E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Diseño y desarrollo de distintos casos de uso de la tecnología Blockchain

Grado en Ingeniería Informática

Trabajo Fin de Grado

Marcos Burdaspar Celada

José Javier Astrain Escola

Pamplona, fecha de defensa

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Contenido

Listado de Figuras

Resumen

En este documento se recoge la memoria del trabajo de final de grado del alumno Marcos Burdaspar Celada en el grado de Ingeniería informática.

Últimamente se ha oído hablar mucho de la tecnología Blockchain que no es otra que la base que sustenta la famosa criptomoneda Bitcoin creada por Satoshi Nakamoto en 2008. Esta tecnología no es otra que la de una base de datos distribuida segura y accesible por todo el mundo.

En el presente trabajo vamos a indagar en las posibilidades que ofrece esta tecnología. Para ello se va a desarrollar tres casos de uso diferentes. Estos casos presentarán tres funcionalidades diferentes que nos podemos encontrar en aplicaciones corrientes .

Para el desarrollo de estos casos de uso se evaluará una serie de plataformas de Blockchain disponibles en la actualidad. Finalmente elegiremos la plataforma de Ethereum para el desarrollo de los casos de uso.

Se planteará un diseño para estos casos de uso, el cual posteriormente será desarrollado, validado y puesto en marcha. Finalmente se analizará tanto comparándolas con las aplicaciones ordinarias, como en el apartado económico.

1 Introducción

En estos tiempos que corren, cada vez están surgiendo nuevas tecnologías que están revolucionando lo que conocemos hoy en día. Una de estas tecnologías es Blockchain. Esta tecnología empezó a ser conocida a través de Bitcoin que fue el primer caso que la popularizó.

El desarrollo de esta tecnología está cobrando mucho auge en la actualidad. Nació con la idea de dar soporte a las criptodivisas pero poco a poco se fue descubriendo toda la capacidad de su potencial. Se está dejando de lado la idea original de criptodivisas y se está evolucionando al desarrollo de otras funcionalidades.

Una de estas funcionalidades que se ha hecho muy llamativa últimamente son los smart contracts o contratos inteligentes. Estos permiten la idea de desarrollar código que funcione directamente en la propia blockchain, pero este tema lo abordaremos más adelante. Otra de las funcionalidades que presenta, son aplicaciones de consenso distribuido.

En este trabajo nos centraremos en explotar el concepto de la primera funcionalidad que he mencionado en el anterior párrafo, los smart contracts. Porque como el nombre indica, esta idea parte del concepto de contrato, pero también ha ido evolucionando. Esta evolución de los contratos inteligentes ha dado lugar a las aplicaciones descentralizadas o como se conocen popularmente, Dapps.

Es este crecimiento en la tecnología de Blockchain lo que me impulsa a realizar este trabajo. Me interesa profundizar en el conocimiento de esta tecnología, sobre todo centrarme en el apartado de los smart contracts y todo lo que ofrecen. Principalmente en el desarrollo de Dapps. Conocer tanto las posibilidades como las limitaciones que ofrece.

1.1 Objetivo

El objetivo principal desarrollo de aplicaciones de software basadas en la tecnología Blockchain. Con esto se pretende averiguar qué prestaciones brinda esta tecnología, cuáles son sus limitaciones si es que tuvieran algunas y también valorar cuales serían los costes que podrían presentar respecto a otras tecnologías más convencionales.

1.2 Alcance

Este proyecto es un trabajo basado en la tecnología blockchain, no en su principal implementación orientada en las criptodivisas, sino en su posterior funcionalidad de la creación de aplicaciones descentralizadas.

En este proyecto se trabaja con esta última funcionalidad que ofrece Blockchain. Se pretende diseñar y desarrollar aplicaciones de uso habitual y ver cómo sería esta implementación junto a esta tecnología.

Esta tecnología la implementaremos en un sistema cloud virtualizado, el cual me ha sido proporcionado por la universidad.

1.3 Estado del arte

Durante estos últimos tiempos se ha empezado a oír la palabra blockchain fuera del contexto de criptomonedas como puede ser bitcoin. Es una arquitectura descentralizada relativamente emergente y con un nuevo paradigma informático.

Recientemente ha atraído el interés de diferentes organismos como gobiernos, entidades financieras o empresas de tecnología. No solo presenta características que permiten el funcionamiento de divisas digitales si no que esto puede ir más allá. Nos permite la ejecución de código propio. También conocido como smart contracts.

En la actualidad existen diversas alternativas que implementan la tecnología de Blockchain de manera que cualquiera la pueda tener disponible. Estas son conocidas como plataformas de Blockchain. Dichas plataformas nos proporcionan la capacidad de implementar nuestras propias Blockchain. Estas plataformas se diferencian unas de otras presentando diferentes características como veremos más adelante.

Generalmente todas las blockchain se pueden clasificar en función de su nivel de acceso como veremos más adelante. Algunas de estas plataformas, nos proporcionan soluciones no solo para implementar una Blockchain. La mayoría de ellas nos proporcionan la capacidad de tener a nuestra disposición criptodivisas ya implementadas para su uso. También algunas de estas alternativas, nos ofrecen lenguajes de programación que permiten el desarrollo de contratos inteligentes de manera más cómoda

En su uso actual, podemos ver cómo cada vez más entidades financieras están apostando por esta tecnología dada su fiabilidad, su rapidez y su capacidad de operar sin intermediarios.

La pregunta que se nos presenta es ¿Qué casos de uso se pueden desarrollar con la tecnología Blockchain que existe actualmente?

En este proyecto se analizan, diseñan y desarrollan diversos casos de uso, que podemos encontrarnos con las tecnologías convencionales, sobre la tecnología Blockchain. El trabajo va a consistir en el estudio, diseño, desarrollo y análisis de software sobre Blockchain.

2 Blockchain

En este capítulo vamos a hablar de todo lo relacionado con Blockchain. Desde donde surgió, a cómo funciona y cómo se encuentra en la actualidad.

2.1 Origen de Blockchain

Lo primero, antes de comenzar nada, vamos a explicar de dónde viene este concepto, el cual va a ser el núcleo de este trabajo. Esta tecnología que tanto está revolucionando el mundo no nació, como todo el mundo cree, con Bitcoin, aunque si fue Bitcoin quien lo popularizó. La idea de Blockchain es un poco más anterior a todo esto, se remonta a 1991. Esta idea fue descrita por dos investigadores Stuart Haber y W. Scott Stronetta [1]. Su idea consistía en algo así como que las marcas de tiempo que había en un documento no se pudieran alterar. Pero tuvieron que pasar casi dos décadas para que finalmente esta idea viera la luz. En enero de 2009 blockchain tuvo su primera aplicación práctica, el lanzamiento de Bitcoin.

2.2 ¿Qué es Blockchain?

Para poder describir este concepto, vamos a empezar por su etimología. Si traducimos su nombre, vemos que literalmente se llama "Cadena de bloques". Con este nombre podemos hacernos una ligera idea de la estructura que puede seguir esta tecnología, pero nos dice muy poco de su funcionamiento. Una definición concisa podría ser la siguiente.

Blockchain es una base de datos distribuida, descentralizada y que solo permite dos tipos de operaciones básicas de base de datos, de escritura y de lectura. Lo que se añade a la cadena es inmutable. Su unidad base es el bloque y cada uno de ellos está unido a su bloque antecesor.

Aunque queda bien condensar la información en unas pocas líneas de texto, vamos a explicar lo que quiere decir cada una de las partes que conforman la definición de arriba. Las bases de datos que todos conocemos presentan generalmente las mismas características. Estas se encuentran alojadas en un servidor o servidores, pero siempre formando parte de una única unidad. Estas bases tienen cuatro tipos de operaciones básicas: seleccionar, insertar, borrar y actualizar. Cuando una aplicación necesita realizar una de las cuatro operaciones mencionadas, se conecta a la base de datos, realiza la operación, devuelve el

resultado obtenido y cierra la conexión. Este sería el método convencional y más utilizado hoy en día.

Ahora vamos a coger la primera parte de la definición que hay arriba. Hay dos palabras que son distribuida y descentralizada, estas vienen a decir que esta base de datos no se encuentra en un punto en concreto, si no que existen diversas copias totales o parciales de la misma base de datos dispersas en diferentes máquinas. Estas bases son alojadas por usuarios, de esta manera no hay una empresa o una entidad que custodie los datos. Posteriormente en la definición pone que solo se pueden realizar operaciones de escritura y de lectura, o lo que es lo mismo, selecciones e inserciones. Esto se debe a la estructura de datos que forman esta base de datos, que es el bloque. Los bloques solo pueden ser añadidos, nunca eliminados o modificados. Aunque ahora estas ideas parece que no tengan ningún fundamento, veremos cómo si lo tienen en los siguientes apartados.

2.3 Como funciona Blockchain

Antes de comenzar con la explicación del funcionamiento de Blockchain, me gustaría explicar un concepto, que se va a repetir mucho de aquí en adelante y es lo que dota a Blockchain de tanta seguridad. Este se trata del término *hash*.

Una función *hash*, es una función que recibe como entrada una cadena de elementos y genera una salida de los mismos elementos que cumple las siguientes condiciones.

- Por cada entrada se genera solo una única salida. En la práctica esto puede suceder, se llaman colisiones y algunos algoritmos antiguos lo padecían. Actualmente los algoritmos que se usan no han presentado ningún caso.
- De una salida es imposible volver a generar su entrada. En la práctica eso se considera muy difícil de conseguir, sobre todo muy costoso a nivel de tiempo, es decir, es poco viable).
- El coste en tiempo de generar la salida a partir de una entrada debe ser muy rápido. Aunque en la práctica esto dependa del procesador donde se ejecute y del tamaño de la entrada, generalmente en los casos más extremos, estamos hablando del orden de segundos.
- Las salidas siempre son de la misma longitud.

Una vez conocido lo que es una función hash, podemos ver todo el potencial que esta nos brinda.

Para poder explicar el funcionamiento de Blockchain, primero hay que definir qué es un bloque. Un bloque es una estructura de datos de tamaño fijo que generalmente está formada por las siguientes partes. La primera, es una referencia al bloque anterior. Generalmente esta referencia suele ser un *hash* de dicho bloque.

Luego guarda lo que vendrían a ser los datos, por ejemplo, en el caso de Bitcoin guarda las transacciones. Finalmente, suele guardar una serie de metadatos como podría ser la marca de tiempo o "timestamp" y un dato llamado *Nonce*. *Nonce* es un acrónimo que hace referencia a la expresión *"number that can be only used once"* o número que solo puede ser usado una vez. Este último número es el método más común para llegar a un consenso con el objetivo de añadir un bloque a la cadena.

Los nodos se encuentran en una red en la cual cuando tienen que añadir la misma información, es decir un nuevo bloque, deben garantizar que se añade el mismo bloque para todos. Esto viene recogido en los algoritmos de consenso. Como la idea es crear una red, la cual no necesite estar validada por una entidad de confianza, es necesario encontrar un método que nos proporcione esta cualidad.

Estos algoritmos de consenso presentan una serie de características como que cada miembro trabaja obviando sus propios intereses, que cada participante tiene la misma importancia dentro de la red y que todos los participantes participan en la elección de un nuevo bloque. A su vez, cada uno de estos algoritmos presenta ciertas diferencias que hacen funcionar la red de manera distinta.

Existen diversos de ellos, pero vamos a mencionar dos de ellos.

El primero es *Proof of work* (Prueba de trabajo). Este fue el primer algoritmo de consenso implementado en Blockchain. Este consiste en que cada nodo debe realizar una tarea costosa que sea fácilmente verificable por el resto de los nodos. Aquí es donde viene el numero *Nonce* que hemos mencionado antes.

El segundo es *Proof of stake* (Prueba de participación). En este caso cada participante realiza una prueba con la idea de que quien más participaciones tenga (Criptomoneda usualmente) sea el que más probabilidades tenga de añadir un nuevo bloque a la cadena. La idea en síntesis sería que premia a los nodos que más interés tienen en el buen funcionamiento de la red.

Como se puede ver cada uno de estos algoritmos presenta una distinción. Comparándolos podemos decir, que el primero presenta una mayor seguridad ya

que presenta pruebas más complicadas a realizar por cada nodo, mientras que el segundo es más rápido a la hora de añadir un bloque.

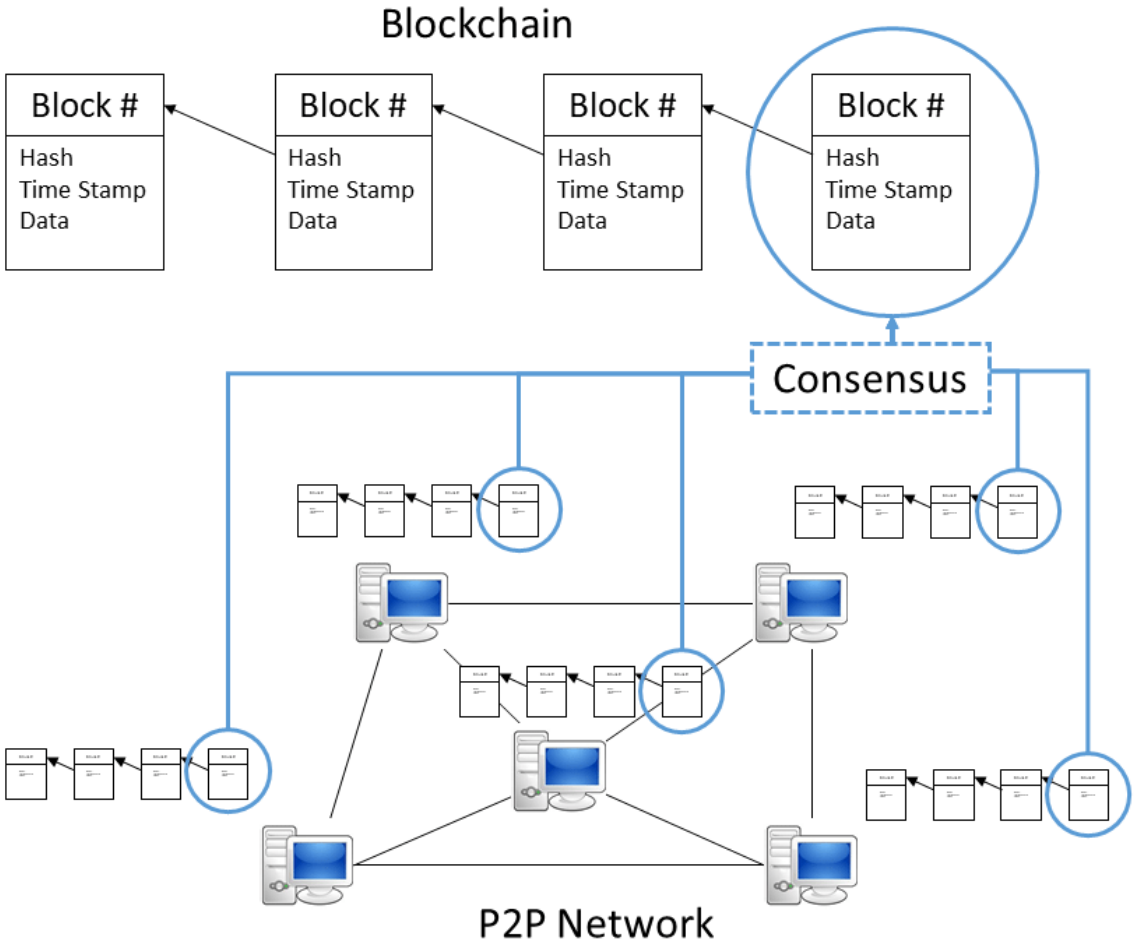


Figura 1: Funcionamiento de Blockchain [2]

En la figura 1 nos encontramos cómo sería la idea en síntesis del funcionamiento de una blockchain. Tenemos una red de punto a punto con diversos equipos a los podemos llamar nodos. Cada nodo almacena una copia entera o parcial de la cadena de bloques. A la hora de añadir un nuevo bloque se establece un algoritmo de consenso y en función de este, se genera el nuevo bloque a la cadena. El nuevo bloque pasa a formar parte de todas las copias de los diversos nodos.

2.4 El origen de Bitcoin

Es muy complicado hablar de Blockchain sin mencionar Bitcoin [3]. Bitcoin fue la pionera en poner a funcionar en todo su esplendor la tecnología de Blockchain, es por ello por lo que me parece interesante dedicarle unos párrafos. También se podrá explicar mejor la tecnología de Blockchain con un ejemplo práctico de ella.

Bitcoin salió a la luz cuando a finales de 2008 se publicó bajo el seudónimo de Satoshi Nakamoto un artículo donde exponía esta idea [4]. Luego en enero de 2009 bitcoin fue introducido como un software de código abierto. El alcance de este proyecto era crear una divisa electrónica para pagos online de igual a igual, sin ningún intermediario o entidad que corroborase dichas transacciones. En él explicaba cómo solucionar la mayor parte de los problemas que existen con las actuales divisas controladas por entidades y bases de datos centralizadas. El método propuesto era nada más y nada menos que Blockchain, eso sí, con sus pequeños matices. En el artículo también explicaba que el método propuesto impedía el caso del doble gasto y además explicaba la seguridad con la que el sistema contaba. También cabe destacar que es un proyecto opensource apoyado por la comunidad. Todo esto es una idea increíble, pero vamos a la parte más interesante, cómo Bitcoin hace uso de la tecnología de Blockchain.

2.5 ¿Cómo funciona Bitcoin?

Como hemos dicho anteriormente, bitcoin utiliza la tecnología de Blockchain para sus propósitos. Por ello, como cuando hemos explicado cómo funcionaba Blockchain, comenzaremos describiendo que es un bloque. En este caso, al ser un ejemplo práctico de la tecnología Blockchain, tenemos una definición más formal de un bloque. Un bloque de Blockchain consta de las siguientes partes: Un número mágico, el tamaño del bloque, la cabecera del bloque, el contador de transacciones y finalmente una lista de transacciones [5]. Pero donde realmente está la gracia es en la cabecera. La cabecera es la clave para validar un nuevo bloque a la cadena. La cabecera a su vez también es una estructura de datos y está formada por los siguientes campos: la versión de la cabecera, el hash de la cabecera del bloque previo, el hash de la raíz de Merkle, el tiempo en que fue concebida la cabecera del bloque, una serie de bits y finalmente el mencionado anteriormente como Nonce [6]. Estos últimos campos que pueden parecer un poco difusos los tratare de explicar junto con todos los campos a continuación.

Actualmente existen muchos algoritmos de hash, todos ellos son de carácter público, es decir en todo momento se conoce que es lo que realizan y aun así es imposible obtener la entrada de la salida. Si nos centramos en el caso de Bitcoin, este usa un algoritmo que se llama SHA-256 [7]. Las siglas vienen de “Secure Hash Algorithm” y el 256 expresa la longitud de salida de la función. En este caso de 256 bits de longitud. Por legibilidad, estos se suelen representar como su valor en hexadecimal. También cabe destacar que Bitcoin cuando emplea la función de hash, lo realiza dos veces, es decir que si denominamos la función hash de x como *“sha256(x)”* bitcoin realizaría lo siguiente: *“sha256(sha256(x))”*.

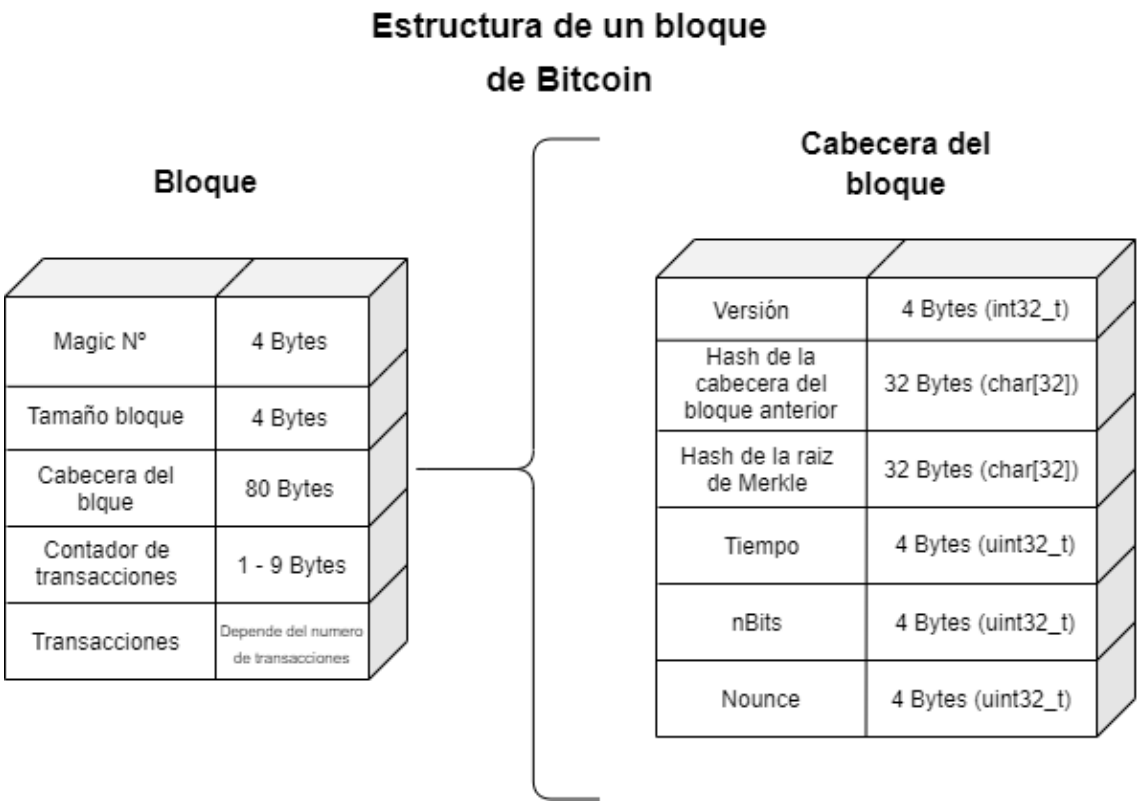


Figura 2: Estructura de un bloque de Bitcoin

2.5.1 El bloque de Bitcoin

- **Número mágico:** Este es el identificador de la red de blockchain de Bitcoin, pero sobre todo indica el comienzo del bloque. Tiene un valor constante y representado en hexadecimal es: 0xD9B4BEF9
- **Tamaño del bloque:** Indica como de largo puede ser el bloque. Hace un par de años, el tamaño de bloque estaba fijado como máximo a 1Mb, pero con el paso del tiempo la comunidad ha cambiado a 2Mb. Esto quiere decir que este valor también puede variar con el tiempo.
- **Contador de transacciones:** Indica el número de transacciones que contiene el bloque.
- **Transacciones:** Es una estructura de datos que contiene todas las transacciones del bloque. El tamaño de este es variable pero siempre respetando el campo del tamaño del bloque establecido.

2.5.2 Cabecera del bloque

- **Versión:** Es un número referido a las actualizaciones de software y al protocolo que se usa.
- **Hash de la cabecera del bloque anterior:** Este guarda ,en formato hexadecimal, la dirección de la cabecera del bloque anterior. Posterior

mente veremos cómo generar el hash de una cabecera. De aquí se puede ver el concepto de la palabra cadena, así es como un bloque apunta a su antecesor.

- **Hash de la raíz de Merkle:** Este es un tipo de estructura de datos creado por Ralph Merkle cuyo funcionamiento explicaré a continuación.

  Para explicar la estructura del árbol de Merkle, primero hay que explicar cómo es una transacción en Bitcoin. Cualquier transacción en Bitcoin está compuesta por dos operaciones: inputs y outputs. O sea, entradas y salidas de bitcoins. Cada una de estas operaciones consta de la salida o entrada pertinente junto con una clave publica que hace referencia a un usuario y la cantidad que va a realizar. En otras palabras, cuando un usuario quiere enviarle bitcoins a otro usuario, se realizan dos operaciones. Una salida al usuario que envía los bitcoins y una entrada al usuario que los recibe.

  Validar todas estas operaciones para que sean añadidas a la cadena sería un trabajo muy costoso. Como solo nos interesa que estas operaciones sean iguales en todos los nodos que forman la red, solo nos hace falta comprobar que sean las mismas entre todos los nodos. Comprobar una por una cada transacción sería algo tedioso y muy poco eficiente. Por ello se usa esta estructura. Esta nos permite resumir todas las transacciones en un valor único y nos permite que la comparación de este valor sea rápida.

  Como hemos hablado anteriormente Bitcoin utiliza la función de hash SHA-256, con lo que cada transacción sería resumida a una salida única de 256bits, que, si los guardásemos en formato hexadecimal, en una cadena de caracteres, se resumiría a 32 bytes. Aun teniendo 32 Bytes por transacción sigue siendo algo arduo de procesar individualmente. Por ello se usa el árbol de Merkle. Siguiendo una estructura de datos tipo árbol, las hojas serían las transacciones, los nodos del siguiente nivel serían los *hashes* de las transacciones. Posteriormente se agruparían de dos en dos y el siguiente nivel sería el *hash* de la combinación de los *hashes* del nodo actual. Así sucesivamente hasta llegar a un único *hash* que sería la raíz del árbol. De esta manera, cualquier alteración de cualquiera de las transacciones daría un *hash* de la raíz diferente, teniendo solo que comparar este único *hash*. La figura 3 ilustra dicha estructura.

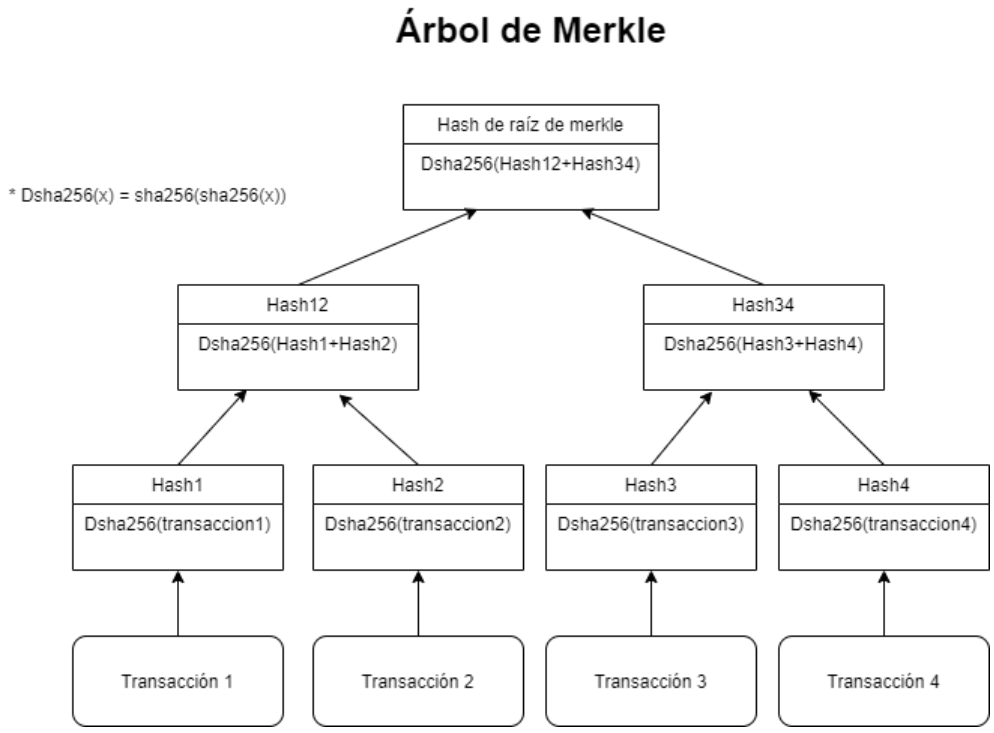


Figura 3: Árbol de Merkle de Bitcoin [8]

* Nótese que como he comentado anteriormente, Bitcoin realiza la función de hash 2 veces.

- **Tiempo:** Este indica la marca de tiempo de cuando es generado el bloque. Este es una representación del tiempo en formato Unix, indica el número de segundos que han transcurrido desde 1970.
- **nBits:** Ese es el nombre que se le ha dado en la página oficial de Bitcoin. Este dato indica la dificultad de minar un bloque. En él se establece de manera compacta un número que mediante la siguiente formula lo transforma a un valor en 256 bits con el que posteriormente se hará una comparación para determinar si el bloque es válido (desarrollare esto más adelante).

$$f(x) = 3ultimosBytes(x) * 256\ \hat{}(primerByte(x) - 3)$$

Ej. $x = 0x181bc330 \rightarrow 0x1bc330 * 256\hat{}(0x18 - 3) = 0x1bc330000000000000000000000000$

- **Nounce:** Este es el famoso número anteriormente mencionado. Este es el número que tiene que ser adivinado para poder añadir el bloque a la cadena.

2.5.3 ¿Cómo se añade un bloque a la cadena?

La pregunta que más recurrentemente me hacía antes de indagar en este tema era, por qué cuesta tanto añadir un bloque a la cadena. Bueno, partimos de que cada nodo que conforma la red de blockchain tiene todos los campos idénticos exceptuando el campo Nounce. Entonces la condición para que el siguiente bloque sea válido para añadir a la cadena es la siguiente.

$$Dsha256(cabeceraBloque) \leq valorCompletoNbits$$

Donde:

$$Dsha256(x) = sha256(sha256(x))$$

$$cabeceraBloque \rightarrow Los\ 80\ bits\ que\ conforman\ la\ cabecera$$

$$valorCompletoNbits \rightarrow valor\ de\ 256\ bits\ de\ nBits$$

Es decir, un bloque es válido cuando el *hash* de la cabecera es menor o igual que el valor descrito en nBits. Vale, ya sabemos cuál es la condición de validación, pero tenemos una incógnita por resolver. Si no sabemos cuál es el valor del Nounce no podemos calcular el hash de la cabecera. Pues no es más que prueba de ensayo y error. Hay que ir probando todas las combinaciones del campo Nounce, calculando el hash del bloque hasta que cumpla la condición descrita. Como se puede suponer, esto no es una tarea fácil y mucho menos cuando compites con el resto de los nodos por añadir el siguiente bloque. Pues este es el motor que sustenta la blockchain que tiene Bitcoin. El primer nodo que encuentre un hash válido, se lleva una recompensa, en este caso, una cantidad de bitcoins concreta. A lo largo del tiempo la cantidad de esta recompensa va variando. Comenzó con una recompensa de 50 BTC (Abreviatura oficial de la divisa de Bitcoin) y cada 210.000 bloques se reduce a la mitad. La idea es que el máximo de bitcoins emitidos sea de 21.000.000. Por otro lado, el campo nBits no es inmutable, este se ajusta para que la generación de nuevos bloques sea siempre de en torno a 10 minutos. Para lograr esto, cada 2.016 bloques se ajustan estos nBits en función del tiempo medio de estos últimos bloques.

Como se puede ver, no es tarea fácil añadir un nuevo bloque a la cadena, pero gracias a este método, la cadena de blockchain puede existir y cumplir los objetivos que esta se proponía.

2.6 Los smart contracts

Bueno, hemos hablado sobre que es Blockchain y sobre Bitcoin que fue el pilar fundamental de esta tecnología. Pero esto no se queda aquí, ni mucho menos.

Después del auge de Bitcoin, empezaron a surgir otro tipo de divisas electrónicas criptográficas: Ethereum (lo explicaremos más adelante), Ripple, Dogecoin, PeerCoin, etc. Cada una de estas divisas traía consigo su propia red de blockchain cada una con su peculiaridad. De estas novedades que presentaban, una de ellas ha cobrado bastante importancia y este es el caso de los smart contracts o "contratos inteligentes".

Como ocurría con la idea de Blockchain, esta idea también llevaba existiendo algún tiempo antes de su fama. En 1996 un criptógrafo llamado Nick Szabo público un artículo donde explicaba la idea de los contratos inteligentes [9]. Como ocurrió con Blockchain, no fue hasta mucho después cuando esta idea se puso en funcionamiento. Sobre todo, porque necesita de una Blockchain para poder existir.

Al igual que pasaba con el origen de Bitcoin, que se quería una divisa cuyos movimientos no estuvieran controlados por terceras partes, los contratos inteligentes siguen el mismo principio. Los contratos convencionales precisan de un tercero para dar validez a ese acuerdo. Los contratos inteligentes van un poco más allá. Son fragmentos de código, más bien, procedimientos almacenados que son añadidos a la cadena solo si ambas o todas las partes que formen el acuerdo están conformes. De esta manera, cuando se cumpla el evento que lanza el contrato este se ejecutará sin terceras partes que lo corroboren. Para intentar expresar un poco esta idea voy a proponer un ejemplo.

Partamos del tema de las herencias. Generalmente cuando una persona fallece, todos sus bienes se traspasan a sus descendientes directos. A la hora de recibir dichos bienes, primero hay que pasar por una serie de procesos de notaría, incluso por Hacienda si no me dejo algo más. La idea es que si ya se sabe cómo se va a realizar el reparto de dichos bienes y todas las partes están de acuerdo, se puede elaborar un contrato inteligente. El contrato estaría compuesto por la división de bienes a sus herederos y el evento que desencadenaría esta acción, es decir el fallecimiento del progenitor. En otras palabras, en el momento de la defunción de la persona, automáticamente sus descendientes recibirían los bienes acordados, sin necesidad de una entidad que lo corroborase.

Como se puede suponer, la idea de los contratos inteligentes presenta una serie de ventajas frente al contrato habitual como podría ser el ahorro en costes de terceras partes y por supuesto su eficiencia respecto al tiempo.

Pero los contratos inteligentes no solo se quedan ahí. Una de las criptomonedas que más ha sabido explotar este concepto ha sido Ethereum, dejando de lado la premisa inicial de "contrato". No solo es un procedimiento para ejecutar algo acordado entre dos partes, sino que expande esta idea a la creación de aplicaciones.

Como hemos hablado en los apartados anteriores la idea de los contratos inteligentes va más allá de la idea de un contrato. Una Dapp es una aplicación distribuida, de ahí la D en el nombre. Es una aplicación almacenada dentro de la red de Blockchain. Por lo tanto, cuenta con la propiedad de la red distribuida y descentralizada de Blockchain. De esta manera una aplicación publicada en la red no está controlada por ninguna entidad.

Al igual que una aplicación o programa que esté conectado a Internet, presenta dos partes el frontend y el backend. El frontend que consiste en la vista que se le presenta al usuario, está formada como cualquier otra aplicación. Si está pensado para el uso por el navegador, esta se presenta como una página web en HTML. La parte interesante se encuentra en el backend, aquí es donde se encuentra la lógica de la aplicación.

Esta lógica de la aplicación es lo que antes hemos mencionado como el Smart contract, se encarga de manejar las peticiones de los frontend. Pero en vez de ser un programa que se ejecuta en un servidor centralizado, es un programa que se encuentra dentro de la red de Blockchain.

También una buena característica a destacar es la capa de almacenamiento de datos. Esta también se encuentra dentro de la red. Por lo tanto, una vez más, cuenta con todas las ventajas de Blockchain. Los datos se añaden a los bloques de la cadena y para evitar accesos indebidos, esta información puede ser cifrada.

2.7 Plataformas de Blockchain

2.7.1 ¿Qué son las plataformas de Blockchain?

En pleno año 2020, ya han pasado más de 11 años desde que Bitcoin salió a la luz. Durante todo este tiempo la tecnología de Blockchain ha ido cobrando más utilidad, hasta tal punto que hay empresas que implementan este sistema.

Como ocurre con otros proyectos informáticos, estos delegan ciertas partes de la aplicación a otros encargados. Véase, por ejemplo, una base de datos, la cual es un programa informático al cual delegas toda la gestión de los datos. En Blockchain ha ocurrido de manera similar. A lo largo de los años han ido

surgiendo diversos proyectos de Blockchain, los cuales implementan su propia red de blockchain, también con sus pequeñas diferencias.

2.7.2 Tipos de plataformas Blockchain

Todos los modelos de cadenas de bloques que se distribuyen en la actualidad, aunque presenten características diversas, se pueden clasificar en tres tipos diferentes: "permissioned", "permisionless" e "hybrid".

2.7.2.1 Blockchain permissioned

Las cadenas de bloques *permissioned* o privadas son aquellas en las que la red de nodos que conforman la cadena de bloques está controlado o limitado. Esto surgió de la entrada de la tecnología Blockchain en empresas y centros privados. Generalmente suele haber un nodo central encargado de dar acceso y permisos al resto de nodos. Es decir, a diferencia de lo que conocemos, si un nodo quisiera añadir contenido a un bloque, o es más, intentar añadirlo a la cadena, primero debería pedirle permiso a un nodo central y este debería concedérselo. Si tuviéramos que resumir las cadenas de bloques privadas, podríamos enumerar los siguientes puntos.

- El acceso a la red de blockchain está restringido únicamente a los nodos que estén autorizados por la unidad central encargada de ello.
- El acceso a las transacciones o al contenido de la cadena es privado.
- Todos los costes de mantenimiento no son llevados a cabo mediante recompensas por criptomonedas. Es la propia entidad la que sustenta económicamente el proyecto.

2.7.2.2 Blockchain permisionless

Como se puede llegar a suponer, fue el primer tipo de blockchain que existió. En este tipo de blockchain cualquiera puede formar parte de la red de nodos que lo conforman y además cualquiera puede consultar las transacciones y los datos que conforman la cadena. También pueden formar parte del desarrollo de su software.

Este tipo de blockchain debe tener en cuenta la seguridad, para impedir que cualquier atacante pueda alterar el buen funcionamiento de esta. Por ello existen los algoritmos de consenso, anteriormente mencionados. También pueden presentar otros ataques como el ataque del 51%, que no es más que el atacante se haga con el 51% de los nodos o el famoso problema del doble gasto, que permitiría a un atacante gastar unas mismas monedas múltiples veces. Intentado resumir otra vez esta idea, expongo a continuación los siguientes puntos.

- Cualquier persona puede formar parte de la red, incluso formar parte del desarrollo de su software.
- Cualquier persona puede consultar las transacciones y el contenido de la cadena.
- El mantenimiento de la red depende del sistema propuesto de la misma, son los propios usuarios los que mantienen a flote la cadena de bloques. Obviamente, sin ser controlados por entidades o empresas.

2.7.2.3 Blockchain hybrid

Como se puede suponer por el nombre, las blockchain *hybrid* o híbridas, son una combinación de las dos anteriores, intentando sacar partido a la mejor parte de ambas versiones. Cogen la parte de las blockchain privadas, las cuales controlan el acceso a la red, para así evitar ataques. Por otro lado, el acceso a la cadena y su información es público, cualquiera puede verlo.

Este tipo de blockchain son útiles para gobiernos o por ejemplo para el sector sanitario. Es decir, no nos interesa que cualquiera pueda validar los datos que van a formar parte de la cadena. Por otro lado, sí que nos interesa que la población o por ejemplo diferentes entidades sanitarias puedan acceder a este contenido sin restricciones. Enumerando una vez más:

- El acceso a la red de nodos es controlado, por un nodo central que da permisos.
- El acceso al contenido de la cadena es público.
- Es parcialmente descentralizado.

2.8 Principales Plataformas

Actualmente existen diferentes alternativas para poder implementar nuestra propia red de Blockchain. Algunas de ellas pensadas para un uso comercial y otras más pensadas para uno uso libre y aprendizaje. A continuación, voy a analizar una serie de plataformas de las cuales una de ellas la adaptare para el desarrollo de este trabajo.

2.8.1 Ethereum

Una de las redes de Blockchain más conocidas después de Bitcoin. Desarrollada por Vitalk Buterin en 2013 [10]. Aparte de ser una red extendida mundialmente, tiene una solución para poder implementar nuestras propias redes de Blockchain. Utiliza una criptomoneda llamada ether que se usa como motor que mueve todos los Smart contracts.

La red que presenta es una red "*permissionless*" y su algoritmo de consenso para determinar la agregación de nuevos bloques es el de "*Proof of work*", aunque se tiene planteado cambiar este algoritmo en un futuro por el de "*Proof of stake*" debido a que este algoritmo es más rápido con respecto al anterior.

El punto más fuerte que presenta esta plataforma radica en que presenta "*Ethereum virtual* machine" (EVM). Esta es una máquina virtual que forma parte del ecosistema de ethereum. Esta se encuentra en todos los nodos de Ethereum. Su funcionalidad principal es interpretar el lenguaje de programación desarrollado por el equipo de Ethereum llamado Solidicity [11]. De esta manera se facilita el acceso a la red, a los recursos y controlando las acciones. También simplifica el desarrollo y diseño de las dapps. Solidicity es un lenguaje de programación de alto nivel que es comparable a javascript o C++. Es un lenguaje orientado a contratos inteligentes, es decir que se enfoca en la creación de Smart contracts.

La organización que lleva a cabo a Ethereum es "*Ethereum Enterprise Alliance*" (EEA) [12]. Es una organización sin ánimo de lucro con más de 250 miembros.

2.8.2 Hyperledger Fabric

Hyperledger Fabric [13] es uno de los proyectos de Hyperledger. Hyperledger es una serie de proyectos de la Fundación Linux. Esta versión es una contribución de IBM y Digital Asset.

Nos encontramos ante una red "*permissioned*", el algoritmo de consenso en este caso es variable, esta plataforma nos da un *framework* para poder implementar el algoritmo que deseemos.

La red que implementemos con esta plataforma, no nos provee de una criptomoneda, pero esta puede ser creada por el desarrollador. En este caso no existe una máquina virtual como en el anterior.

Esta plataforma ha sido enfocada para el uso empresarial, fundamentalmente por la parte de controlar el acceso a la red como hemos visto antes.

2.8.3 Hyperledger Sawtooth

Hyperledger Sawtooth [14] es también uno de los proyectos de Hyperledger. Este al igual que el anterior, forma parte de la Fundación Linux. También nos encontramos con que es una contribución de IBM y Digital Asset.

Nos volvemos a encontrar una plataforma "*permissioned*". Esta vez tenemos un algoritmo de consenso diferente. Tenemos el algoritmo de consenso PoET (Proof of Elapsed Time). Este algoritmo permite a esta plataforma integrarse con soluciones de seguridad del hardware, lo que hace que se conozcan como entornos de ejecución confiables.

Aunque estos son los caracteres generales de la plataforma. Esta también presenta un rasgo muy distintivo, es modular. Esta modularidad permite que las entidades o empresas presenten sus políticas respecto al uso de la cadena de bloques. Es decir que pueden elegir las reglas de transacción, permisos y algoritmos de consenso.

2.8.4 Ripple

Esta plataforma [15] nació con el objetivo de conectar proveedores de pago, intercambio de activos en el mundo digital y demás transacciones a través de una red de blockchain sin comisiones de por medio. Se creó un activo digital llamado "Ripple" que ahora forma parte del mundo de las criptomonedas como Ether y Bitcoin.

Se podría decir que presenta un tipo de blockchain híbrida y tiene un algoritmo de consenso de votación probabilística para alcanzar el consenso entre los nodos.

Empresas como Santander y American Express están probando el potencial de esta plataforma para poder integrarlo en procesos de pagos ya existentes para convertirlos en más seguros y rápidos.

2.8.5 Hedera Hashgraph

Nos encontramos con una plataforma [16] que presenta un algoritmo de consenso distribuido novedoso que no se trata de uno con una prueba de trabajo pesada. Las características que presenta esta plataforma se resumen en tres: es segura, rápida y justa.

Esta es una plataforma que implementa una red de blockchain permissioned. Presenta una solución para el desarrollo de contratos inteligentes de tal manera que se asocian a unos árbitros los cuales pueden editar este contrato pudiendo así añadir nuevas funciones.

2.9 Agentes que impulsan Blockchain

En este apartado me gustaría hablar de los agentes que impulsan a Blockchain. Durante estos últimos años han ido surgiendo instituciones u organizaciones que

han ido fomentando el uso de Blockchain. Me gustaría mencionar alguna de ellas, tanto a nivel nacional como a nivel internacional.

2.9.1 Alastria

Alastria es una asociación sin ánimo de lucro [17]. Su intención es fomentar la economía digital a través de esta tecnología descentralizada. A esta entidad nacional, están asociadas diversas empresas importantes en España como pueden ser BBVA, Acciona o Telefónica.

Alastria, hoy en día, está formada por un 45% de PYMES, 12% de grandes empresas y un 43% de otras instituciones. Presenta una red pública permisionada. Otro dato muy relevante de esta asociación es que la Universidad Pública de Navarra forma parte de ella también.

2.9.2 BLUE

Como sus siglas indican, Blockchain Universidades Españolas, es una red de Blockchain impulsada por universidades españolas [18]. La idea es comenzar a desarrollar y desplegar servicios en ella.

En la implementación contamos con Hyperledger Fabric como plataforma. Actualmente cuenta con tres módulos: Registro de titulaciones, Validación / Verificación e implementación de consultas y participación de titulaciones.

2.9.3 R3 Corda

R3 es una empresa que lidera un entorno de más de 300 empresas de origen internacional [19]. R3 junto a su entorno, desarrolla la plataforma Corda [20].

Se diferencia por presentar una nueva solución para el mundo empresarial. Toda la red global de Blockchain trabaja equitativamente por diseñar y desarrollar aplicaciones para el comercio y las finanzas utilizando el software de Corda.

3 Desarrollo de la plataforma de Blockchain

En este apartado vamos a presentar los casos de uso a realizar y analizar qué tecnologías se usarán para llevarlos a cabo. No solo hablaremos de qué plataforma de Blockchain vamos a utilizar, si no de cómo se diseñan y se desarrollan los casos de usos. También hablaremos de la parte del cliente.

3.1 Casos de uso propuestos

3.1.1 Red social tipo Twitter

En la actualidad, las aplicaciones que más se usa en el día a día, son las redes sociales. La idea de ver si una red social podía ser implementada dentro de una red de blockchain es un buen caso de uso para ver si esta soporta las necesidades de la red social.

También hay que mencionar que contaríamos con las ventajas de una blockchain, estaríamos hablando de una red social distribuida sin una entidad que estaría detrás de ella. Lo que probablemente nos hace pensar que ya no haría falta vender tu privacidad a cambio de poder disfrutar de estos servicios.

La red social propuesta va a ser similar a la red social de Twitter, o por lo menos parte de sus funcionalidades. Los requisitos que va a implementar este caso de uso son los siguientes.

- Cada usuario tiene que ser identificado unívocamente en la aplicación.
- Cada usuario puede escribir varias cadenas de caracteres (Tweets).
- Cada usuario puede leer todos sus tweets.
- Cada usuario puede seguir a otros usuarios.
- Cada usuario pude leer los tweets de los usuarios que sigue.

3.1.2 Diario privado o Cuaderno de bitácora

Uno de los programas más antiguos que existen son los editores de texto o bloc de notas que te permiten guardar archivos de textos. Estos programas remplazaron en mayor o menor medida los cuadernos normales donde la gente escribía lo que les sucedía o se les pasaba por la mente.

Aprovechándonos de la tecnología Blockchain y su descentralización, la idea de este caso de uso es la de implementar un cuaderno de bitácora que te permita escribir contenido dentro de él, que a su vez este se encuentre distribuido a lo largo de la red y que además este sea privado. Los requisitos que va a implementar este caso de uso son los siguientes.

- Cada usuario tiene que ser identificado unívocamente en la aplicación.
- Cada usuario puede escribir entradas en el diario (cadena de texto).
- Cada entrada estará acompañada de su fecha de publicación.
- Cada entrada se guardará cifrada (AES) mediante una contraseña.
- Las entradas se descifrarán (AES) mediante una contraseña.

3.1.3 Juego online, tres en raya

En la industria del software, los videojuegos son uno de los sectores que más beneficios aportan. Este caso de uso trata de implementar el soporte para un juego online. Se trata del conocido tres en raya. Con este caso de uso podremos explotar en mayor medida el potencial de plataforma de Blockchain escogida ya que no solo se va a limitar a la funcionalidad de CRUD (crear, leer, actualizar y eliminar) que presentaban los otros casos de uso.

Para este caso de uso los requisitos previstos son los siguientes:

- En cada partida solo podrán jugar dos jugadores.
- Solo se puede realizar una partida simultáneamente.
- Cada jugador tiene que marcar una posición en su turno.
- Cuando no es su turno debe esperar a que lo sea.
- La partida finalizara cuando tres posiciones adyacentes sean del mismo jugador.

3.2 Plataforma elegida

Después de haber analizado las diferentes propuestas mencionadas anteriormente, vamos a seleccionar una de ellas para la realización del trabajo.

Tras compararlo con las diferentes alternativas presentadas con anterioridad, he decidido que voy a utilizar la plataforma de Ethereum. Esta plataforma carece de un carácter puramente empresarial y por lo tanto tiene un carácter más general. Es por ello que nos brinda una mayor libertad a la hora de interaccionar con la cadena de bloques.

Dado que el origen de Ethereum es anterior a la de las plataformas mencionadas previamente, este cuenta con una comunidad más grande. El hecho de tener una mayor comunidad activa facilita la recopilación de información para interactuar con esta plataforma.

Al tener toda esta presencia, han surgido herramientas que simplifican el desarrollo de los proyectos. Como veremos más adelante, estas herramientas nos permiten crear nuestra propia red de Blockchain de desarrollo y además poder utilizar cuentas de monederos virtuales con los que poder interactuar.

Como es una red con muy poca personalización, está ya viene configurada. Entre otras cosas viene establecido el algoritmo de consenso de "*Proof of work*" el cual será más que suficiente para llevar acabo el desarrollo de este proyecto. Además, no es necesario establecer un token o criptodivisa propio, ya que viene configurado el suyo propio *Ether*.

Pero el factor más distintivo que me ha hecho decantarme por elegir esta plataforma, es el siguiente. Aunque muchas otras plataformas presentan soluciones fáciles y cómodas para la creación de aplicaciones distribuidas, esta presenta un rasgo muy singular. Esta plataforma presenta un lenguaje orientado a contratos, es decir un lenguaje creado única y exclusivamente para la creación e interacción con un tipo de estructuras llamadas contratos. Este lenguaje se llama Solidity y aunque aún no se encuentre acabado, a día de hoy sigue evolucionando, facilitando el desarrollo de contratos inteligentes.

Este lenguaje no sería posible si no existiera un intérprete capaz de analizarlo. Este intérprete es la *Ethereum Virtual* Machine (EVM). Este interprete, se encuentra en cada nodo que forma la cadena de bloques y es el encargado de hacer funcionar nuestros contratos.

3.3 Entorno de desarrollo

3.3.1 Parte del servidor/Blockchain (Backend)

3.3.1.1 Nuestra propia Blockchain

En el caso de Ethereum, si quieres publicar tu Smart contract en la red, este sí que cuesta ether para poder publicarlo. Afortunadamente, tenemos una solución para poder desarrollar contratos inteligentes libremente. Vamos a utilizar un programa que se va a encargar de crear nuestra propia Blockchain personal. Este programa se va a encargar de generar los bloques e ir creando la cadena, de esta manera nos va a abstraer del funcionamiento interno de la red de Blockchain y así poder centrarnos en el desarrollo de las dapps.

El software que se ha mencionado anteriormente se llama Ganache. Es un software desarrollado por el equipo de Truffle [21]. Este software aparte de desplegar nuestra propia Blockchain, nos provee de unos monederos con ether (solo para nuestra red) para poder publicar los contratos. También nos facilita la

tarea de despliegue y testeo de los contratos. Cuenta además con una versión con interfaz gráfica para poder visualizar todos los parámetros de la red creada de forma más cómoda.

Para instalar este software nos dirigiremos a su página web para realizar la descarga del instalador. La página de descarga es la siguiente "https://www.trufflesuite.com/ganache" como se puede apreciar en la figura 4.

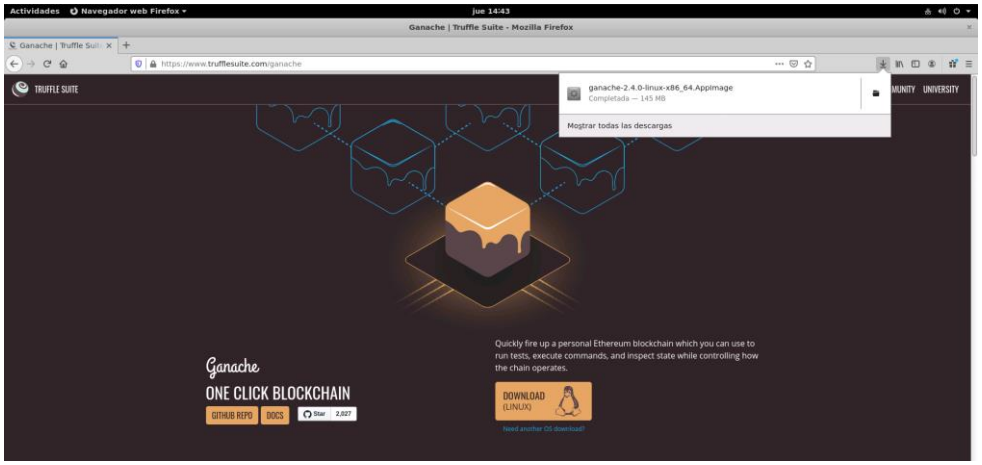


Figura 4: Botón de descarga de Ganache

Una vez descargado el instalador con el nombre de archivo: *ganache-2.4.0-linux-x86_64.AppImage*, procederemos a instalarlo. En mi caso el archivo se encuentra en el directorio de descargas. Iniciaremos un terminal y nos dirigiremos al directorio de descargas para ejecutar los siguientes comandos.

```
$ chmod 500 ganache-2.4.0-linux-x86_64.AppImage
$ ./ganache-2.4.0-linux-x86_64.AppImage
```

Primero le concedemos al archivo permisos de ejecución y después lo ejecutamos. Se nos abrirá una ventana con los pasos a seguir como se ve en la figura 5.

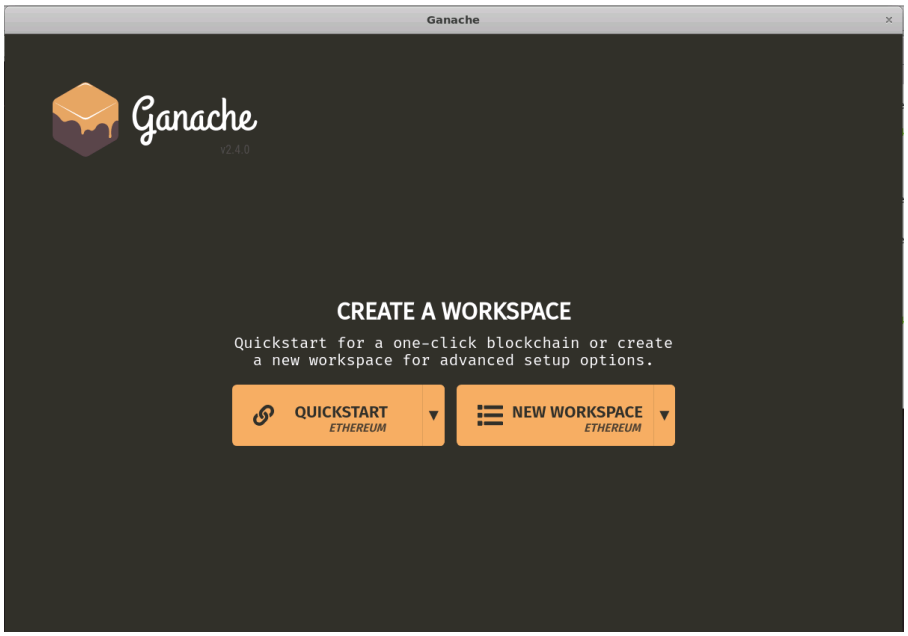


Figura 5: Menú de arranque de Ganache

Seleccionaremos la opción de *quickstart* y se nos desplegara el menú principal de la aplicación.

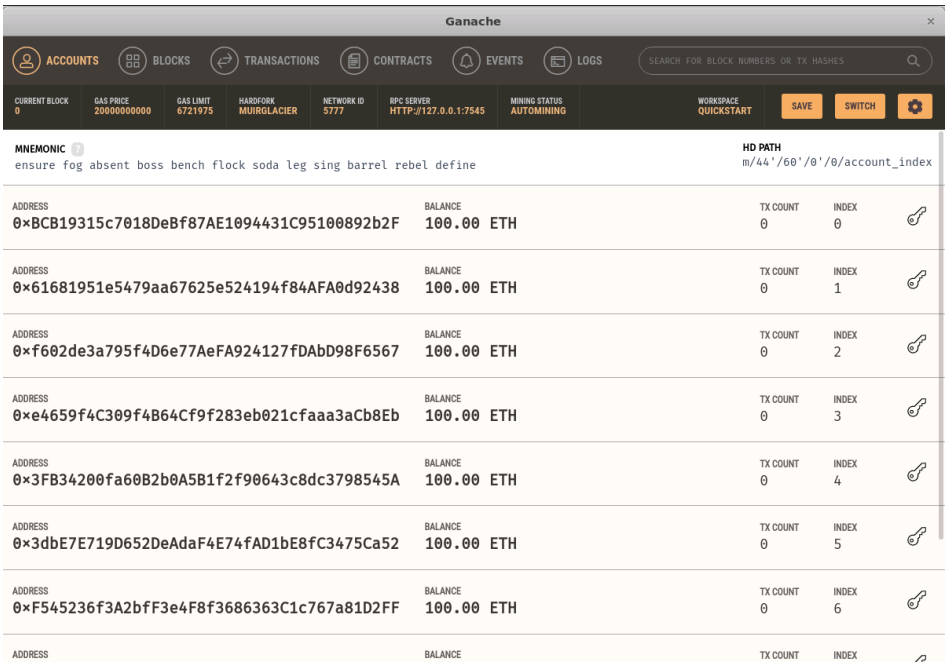


Figura 6: Menú principal de Ganache

En este menú que vemos en la figura 6 podemos apreciar varias cosas a destacar. En la primera pestaña vemos como tenemos unas diez cuentas ficticias con 100 de ether cada una. Usaremos estos ether para publicar los contratos. En la siguiente pestaña vemos una lista con los bloques que se han generado, después

una con las transacciones y otra con los contratos publicados. Todas estas opciones las iremos viendo conforme las vayamos necesitando más adelante.

3.3.1.2 Node Package Manager Y Truffle Framework

Una vez tenemos ya Ganache instalado, tenemos que instalar unas dependencias para poder comenzar con el desarrollo de los contratos inteligentes. Necesitamos instalar un framework de Truffle para poder desarrollar nuestros contratos con el lenguaje de programación que nos brinda Ethereum, Solicity. Este framework nos permitirá validar nuestro contrato además de poder desplegarlo dentro de la red.

Resulta que, para poder instalarlo, necesitamos usar el gestor de paquetes de Node (Node Package Manager). Así que previamente hay que instalar NPM para poder instalar el framework de Truffle. Ejecutaremos los siguientes comandos para instalar el gestor de paquetes de Node.

```
$ sudo apt-get install curl
$ curl -sL https://deb.nodesource.com/setup_12.x | sudo -E bash -
$ sudo apt-get install nodejs
```

Primero instalaremos curl, si este no estaba ya previamente instalado. Curl es un proyecto de software orientado a la transferencia de archivos con diferentes protocolos conocidos. De ahí obtenemos del repositorio oficial de node.js los archivos para instalar node [22]. Nos bajamos la versión 12, que es la versión más estable. Una vez hecho esto instalamos nodejs como la mayoría de los paquetes que se instalan en Debian, con apt. Finalmente comprobamos que se ha instalado correctamente como se ve en la figura 7.

```
marcos@blockchain:~/Descargas$ node -v
v12.16.3
marcos@blockchain:~/Descargas$ npm -v
6.14.4
```

Figura 7: Versiones de Node y NPM

Una vez que ya tenemos instalado el gestor de paquetes de node, procederemos a instalar el framework de Truffle. Ejecutamos el siguiente comando para hacerlo.

```
$ sudo npm install -g --unsafe-perm=true --allow-root truffle
```

Ejecutamos el comando con sudo ya que tiene que acceder a ciertos directorios en los cuales necesita permisos. Los argumentos *–unsafe-perm=true* y *–allow-root* los he tenido que poner ya que me daba un error de acceso, puede que se deba a estar implementando esto en la máquina virtual.

3.3.2 Parte del cliente (Frontend)

3.3.2.1 Metamask

La parte del cliente va a estar realizada mediante una interfaz web. Para poder trabajar con los contratos inteligentes, es necesario identificarnos y por ello en necesario instalar este plugin para el navegador.

Al ser la idea convencional de que un contrato está formado por varias partes que llegan a un acuerdo, es necesario identificar dichas partes. En este caso, aunque el contrato sea una aplicación que nada tiene que ver con interactuar con diferentes partes, es necesario identificarse para poder usar la infraestructura creada. Para identificarnos usaremos una de las cuentas que nos facilita Ganache. También hay que mencionar que este plugin para el navegador tiene como propósito garantizar las transacciones de ether, es decir, funciona como un monedero virtual. Este plugin se usa para interactuar con la red de ethereum principal, pero en nuestro caso lo configuraremos con la red local creada por Ganache.

En la máquina virtual el navegador que vamos a utilizar es Mozilla Firefox, así que para instalar este plugin vamos a realizar lo siguiente. Nos dirigimos a la página oficial de plugins de Firefox [23] y presionamos el botón de agregar a Firefox, como se muestra en la Figura 8.

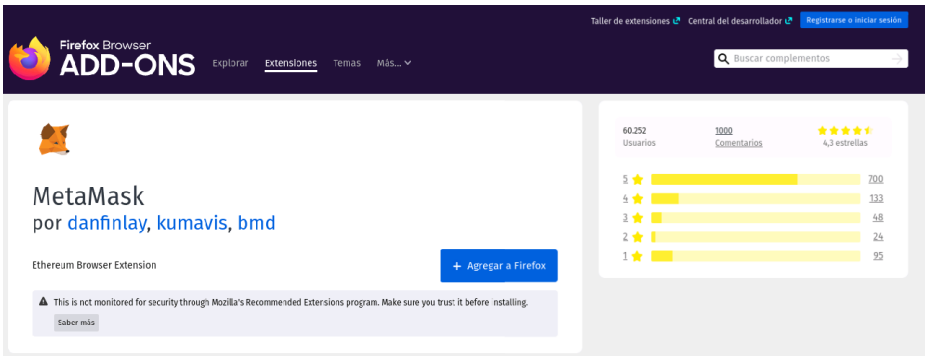


Figura 8: Botón de agregar a Firefox de Metamask

Ahora vamos a configurar este plugin. Lo iniciamos ya sea dándole al icono que suele aparecer arriba a la derecha o bien nada más instalarlo nos aparecerá un botón para empezar. Lo primero que nos pedirá es que asignemos una contraseña para poder crear el monedero, en mi caso he dejado que Firefox me genere una contraseña aleatoria y la guarda él. Posteriormente nos dará una frase con una serie de palabras aleatorias, que nos preguntará si nos olvidamos de la contraseña.

Una vez llegados aquí tenemos ya el menú principal. Ahora vamos a configurar Metamask con nuestra red de Blockchain local. Para ello abriremos un desplegable que se encuentra arriba a la derecha. En él se nos mostrará la red principal de Ethereum, otras secundarias y finalmente una opción que pone RPC personalizado. Es en esta última donde tenemos que clicar. Se nos mostrará el siguiente formulario y lo rellenaremos con los datos que aparecerán a continuación en la Figura 9.

Figura 9: Formulario para añadir una nueva red Ethereum a Metamask

Simplemente le hemos dado un nombre a la red que vamos a añadir y posteriormente le hemos puesto la URL para que se conecte a la nuestra red local de Blockchain. La url la hemos obtenido del campo RPC server que nos encontramos en el menú principal de Ganache. Esto se puede apreciar en la Figura 9.

3.3.2.2 Frameworks para la interacción con los Smart Contracts

La parte del cliente estará desarrollada como una página web, eso quiere decir que necesitaremos usar HTML, CSS y JavaScript. Aun con todo esto, no podremos relacionarnos con la red, necesitaremos de algo más. Para ello será necesario añadir dos librerías de JavaScript. Estas ya se nos añaden al inicializar el proyecto como veremos más adelante.

Estas dos librerías son web3.js y truffle-contract.js. La primera es la que interactúa sobre todo con Metamask y nos permite obtener la autentificación del usuario. La segunda es la que nos permite interaccionar con los contratos cuando realizamos alguna llamada a la red.

3.3.2.3 Framework Bootstrap

He añadido al proyecto también, el framework gratuito de Bootstrap. Este, hará que la página web se vea un poco mejor que por defecto. Para poder tenerlo hace falta la librería de JQuery de JavaScript que también nos será de utilidad. Para

añadirlos a la parte del cliente, solo es necesario bajarse los archivos .css y .js de la página oficial de bootstrap [24] y copiarlos en los directorios correspondientes dentro del directorio src. También he añadido Bootstrap al proyecto debido a que estoy familiarizado con él y no me supone un gran esfuerzo usarlo.

3.3.3 IDE (Entorno de desarrollo integrado) a utilizar

Finalmente, vamos a utilizar un IDE gratuito, para poder desarrollar nuestro código de manera más cómoda. El IDE que voy a elegir para es Visual Studio Code, por dos motivos. El primero es porque estoy familiarizado con él por haberlo usado con anterioridad y el segundo es porque este cuenta con una extensión de Solidity que no todos los IDE cuentan con ella.

Para instalarlo, en mi caso en la máquina virtual de Ubuntu 18.04, hay que realizar lo siguiente. Nos dirigimos a la página de descargas de Visual Studio Code "https://code.visualstudio.com/Download". Una vez ahí nos bajamos la opción que pone *.deb* y esperamos a que se descargue. Nos dirigimos al directorio donde se halla descargado y ejecutamos el siguiente comando, posteriormente verificamos que se ha instalado correctamente como se aprecia en la Figura 10.

```
$ sudo apt install ./code_1.45.1-1589445302_amd64.deb
```

```
marcos@blockchain:~/Descargas$ code -v
1.45.1
5763d909d5f12fe19f215cbfdd29a91c0fa9208a
x64
```

Figura 10: Verificación de la instalación de VSC

Una vez instalado, instalaremos la extensión de Solidity. Iniciamos Visual Studio Code y vamos al apartado de Extensiones y buscamos la extensión de Solidity. Una vez ahí le damos a instalar como se aprecia en la Figura 11.

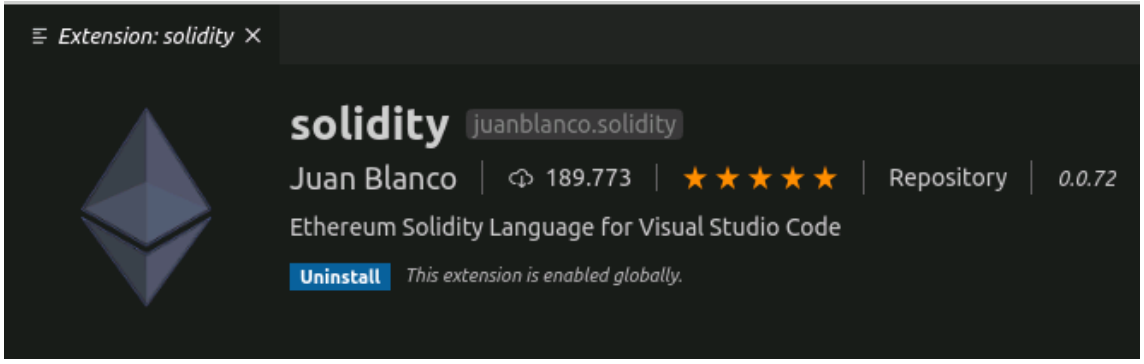


Figura 11: Instalación de la extensión de Solidity

También hay que comentar que para la realización del frontend, vamos a desarrollar páginas webs. Por ello, este IDE también nos facilitará el desarrollo de estas.

Una vez tenemos instaladas todas las dependencias necesarias vamos a comenzar con el desarrollo de los diferentes casos de uso.

4 Diseño y desarrollo

4.1 Primer caso de uso: Red Social (Twitter)

4.1.1 Backend

Para comenzar con el desarrollo, no vamos a partir de un proyecto en blanco, vamos a aprovecharnos de una de las funcionalidades que nos ofrece Truffle. En este framework, existen una serie de proyectos preparados para que la gente se los baje y pueda trabajar con ellos. Nosotros nos vamos a aprovechar de esta funcionalidad para inicializar nuestro proyecto. Este proceso lo realizaremos para inicializar los demás proyectos de los otros casos de uso.

Para ello vamos a realizar los siguientes comandos.

```
$ mkdir twitter
$ cd twitter
$ truffle unbox pet-shop
```

Primero crearemos el directorio de nuestro proyecto y una vez dentro ejecutamos el comando de Truffle para bajarnos ese proyecto.

Una vez hemos bajado el proyecto, podemos observar en la Figura 12 una serie de directorios en su interior a los cuales vamos a tener que ir a parar más adelante. De modo rápido, vamos a desatacar 3 de ellos. Uno de ellos es el directorio contracts; aquí se encontrará el código que generemos en Solidity, la parte que se encargará de la lógica de negocio de la aplicación. Otro de ellos es el directorio migrations. En la parte del servidor se ejecutará sobre Node.js, por ello es necesario desplegar los contratos en la red de Blockchain. En este directorio tenemos el código en javascript para desplegar los contratos en la red. Para mencionar al último, tenemos el directorio src, en el que se encuentra el código del frontend que será el cliente ejecutado por el navegador.

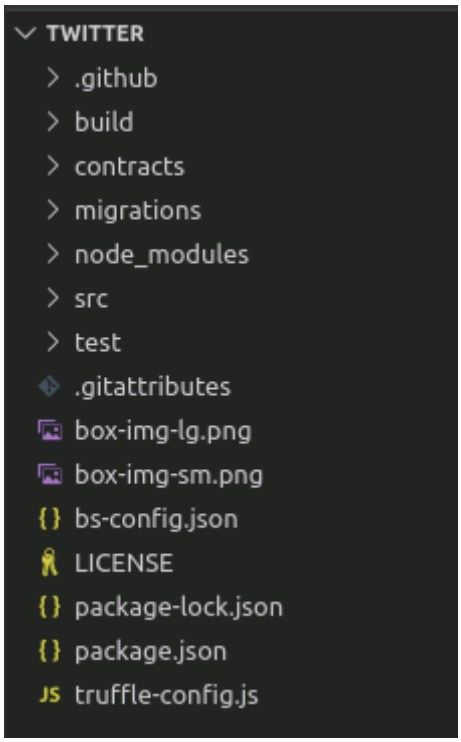


Figura 12: Contenido del directorio de Twitter

Ahora vamos a comentar como se realizaría el despliegue y puesta en marcha del proyecto. Para compilar un contrato usaremos el siguiente comando.

```
$ truffle compile
```

Este comando nos dirá si el contrato escrito en Solidity ha podido procesarse sintácticamente. Aunque según he podido observar, el IDE con la extensión ya hace una pre-compilación avisándote hasta de los warnings. Incluso te avisa si alguna característica solo está en modo experimental y te dice que debes poner una cabecera especial para ello. Con lo que la extensión para este IDE ha sido de gran ayuda.

El siguiente comando nos permitirá publicar el contrato dentro de la Blockchain, para ello previamente, debemos tener iniciado Ganache, que como hemos mencionado anteriormente esta será nuestra Blockchain local. Hay que tener en cuenta que estamos en una red de Blockchain de desarrollo, por lo que no nos costará Ether real publicar nuestro contrato, pero una vez ejecutado el comando siguiente, este será publicado y cobrado.

```
$ truffle migrate
$ truffle migrate –reset
```

El segundo comando lo usaremos cuando hemos modificado nuestro contrato y queremos que este se modifique en la red. Teniendo en cuenta que no es una modificación, borra el primer contrato desplegado y añade el nuevo modificado. Cuando se ejecuta este comando, nos aparece un "recibo" con todo lo que ha sucedido y el coste que nos ha llevado. Posteriormente enseñaré una imagen de esto.

El último de los comandos que vamos a necesitar así de carácter general va a ser el que nos permita desplegar nuestro cliente en el navegador. Concretamente este comando nos lo proporciona Node.js que hace de servidor el cual lanza el código HTML, JS y CSS al navegador del usuario. El comando es el siguiente.

```
$ npm run dev
```

Una vez que tenemos las herramientas adecuadas para poder llevar a cabo nuestro proyecto, vamos a comenzar con él.

El código que he desarrollado en Solidity para modelar el comportamiento que debería tener la aplicación es el siguiente:

```
pragma solidity >0.4.2;

contract Tweet{

    mapping (address => mapping(uint => string)) tweets;
    mapping (address => uint) numTweets;
    mapping (address => mapping(uint => address)) seguidos;
    mapping (address => uint) numSeguidos;

    function addTweet(string memory tweet_str) public{
        numTweets[msg.sender]++;
        tweets[msg.sender][numTweets[msg.sender]] = tweet_str;
    }

    function getNumberTweets() public view returns (uint) {
        return numTweets[msg.sender];
    }

    function getOwnTweets(uint i) public view returns (string memory){
        return tweets[msg.sender][i];
    }

    function seguir(address guy) public {
        numSeguidos[msg.sender]++;
        seguidos[msg.sender][numSeguidos[msg.sender]] = guy;
    }
```

```
    function getNumSeguidos() public view returns(uint) {
        return numSeguidos[msg.sender];
    }

    function getSeguidos(uint i) public view returns(address){
        return seguidos[msg.sender][i];
    }

    function getNumTweetAddress(address guy) public view returns(uint){
        return numTweets[guy];
    }

    function getTweetSeguidos(address n, uint i) public view returns(string memory) {
        return tweets[n][i];
    }
}
```

Ahora voy a explicar un poco el diseño de este contrato. Primero he de decir que las funcionalidades que he implementado han sido la capacidad de poder publicar una cadena de texto (Tweet) y que esta se vea y la capacidad de ver las cadenas de texto publicadas por los usuarios a los que sigues.

Vamos a comenzar hablando de toda la parte de usuarios. Esta se ha realizado con el plugin Metamask. Con este plugin, podemos acceder a nuestro monedero virtual, en el cual están nuestras cuentas de ethereum. Estas cuentas están identificadas con una cadena de 42 caracteres que identifican con un código en hexadecimal al usuario. Un ejemplo de código de cuenta sería el siguiente.

0x742d35cc6634c0532925a3b844bc454e4438f44e

De esta manera, todo el tema de autentificación de usuario, lo hemos delegado a este componente, de tal manera que nosotros podemos obtener esta cadena en hexadecimal para poder identificar unívocamente al usuario.

Dicho esto, vamos a empezar a analizar cómo funciona el contrato. En Solidity, aparte de existir las variables comúnmente conocidas en casi todos los lenguajes de programación, tenemos un tipo de variable llamado address, el cual sirve para guardar el código mencionado en el párrafo anterior.

Al principio del contrato podemos ver una sentencia llamada mapping, esta sería como un array asociativo. Es decir, un conjunto de pares de clave valor. El primero que nos encontramos es el de tweets. Este es un doble mapping, se podría decir que a cada usuario se le asocia una lista de strings. Lo he hecho de esta manera

porque Solidity da problemas al declarar arrays de Strings, parece que es una característica que añadirán en futuras versiones. En esta estructura, se van a guardar el contenido de los "Tweets" que escriban los usuarios.

Después tenemos otro mapping, en este caso asocia a cada usuario, el número de "Tweets" que ha escrito. Guardo esta información debido a que en las estructuras de mapping, no se puede saber el número de elementos que contienen. Es decir, así como en otros lenguajes de programación, que existe un atributo o método llamado length, en este no existe. Esta variable se llama *numTweets*.

El último mapping asocia a cada usuario con el número de personas a las que sigue. Repito que guardamos esta variable para poder utilizar el número de seguidos por cada usuario.

Las siguientes sentencias que nos encontramos en el contrato, son funciones. La primera de ellas nos permite añadir un tweet. Para ello voy a explicar una parte que se repite a lo largo de las funciones. Esta parte es *msg.send.* Esta llamada devuelve el código de la cuenta en hexadecimal del usuario que ha hecho la llamada a la blockchain. Por ello, podemos ver que en función addTweet se hace lo siguiente. Primero aumentamos en numTweets en uno al usuario que ha entrado en el contrato. Posteriormente, con el número de tweets y el usuario, añadimos el tweet en el mapping.

Otra cosa que se puede observar es que, algunas veces, delante de la palabra string se ve una palabra reservada llamada memory. En Solidity, cuando se declaran las variables es preciso emplear una de las dos palabras reservadas *memory* o *storage*. Esto es así, porque como he mencionado antes, publicar un contrato cuesta ether. La palabra reservada *memory*, hace que esa variable no persista en la memoria de la máquina virtual mientras que *storage* sí que lo hace, también repercute en el coste de la publicación del contrato.

La segunda función nos devuelve el número de tweets que ha realizado un usuario. Para ello accede a el mapping de *numTweets*.

La siguiente, *seguir()*, pasado como parámetro el índice del tweet, devuelve el contenido del tweet de esa posición del usuario que ha llamado a esta función. Una opción mejor que me hubiera gustado hacer, hubiera sido traer toda la estructura del mapping, dado el código del usuario, pero esto no ha podido ser debido a que Solidity presentaba esta limitación en la devolución de datos de las funciones.

Después nos encontramos con la función seguir. Esta, dado un código de usuario, se añade a la lista de seguidos de la persona que ha llamado el contrato.

La siguiente devuelve el número de personas que sigue la persona que ha llamado al contrato.

La sucesiva, *getSeguidos()*, dado un número que actúa de índice dentro del *mapping*, nos devuelve el código de la persona que se encuentra en la posición del número dado, a la cual sigue el usuario que ha llamado al contrato.

En la función posterior, dado una cuenta de usuario, esta nos devuelve el número de personas a las que sigue.

En la última función, dado el código de un usuario y un número, esta nos devuelve del usuario introducido y en la posición indicada.

Comentar una vez más que igual el código puede que se vea un poco engorroso, pero debido a las limitaciones con las que me he podido encontrar con este lenguaje me he visto obligado a realizarlo de esta manera, que funciona correctamente.

4.1.2 Testing

Como hemos visto anteriormente, la publicación de un contrato cuesta una cierta cantidad cada vez que se produce, por lo que, en este tipo de software, la parte de verificación y validación cobra una mayor importancia.

El framework de Truffle, nos proporciona una solución para poder realizar pruebas a nuestro contrato. Las pruebas que desarrollemos para un contrato se encontrarán dentro del directorio *./test* del proyecto. Los tests son archivos en javascript, concretamente usando Node.js.

Una vez que tenemos los tests, para ejecutarlos, tenemos que realizar el siguiente comando dentro del directorio de nuestro proyecto.

```
$ truffle test
```

Una vez hemos visto cómo funcionan los test a los contratos con truffle, vamos a ver el test desarrollado para este contrato.

```
var Tweet = artifacts.require("./Tweet.sol");

contract("Tweet", function(accounts) {
  var tweetInstance;
```

```
  it("TestAddTweet", function() {
    return Tweet.deployed().then(function(instance) {
        tweetInstance = instance;
        return tweetInstance.addTweet("Mensaje de prueba", { from: accoun
ts[1] });
    }).then(function() {
        return tweetInstance.getNumberTweets({ from: accounts[1] });
    }).then(function(num) {
        assert.equal(num, 1);
        return tweetInstance.getOwnTweets(1,{ from: accounts[1]});
    }).then(function(contenido) {
        assert.equal(contenido, "Mensaje de prueba");
    });
  });

  it("TestSeguir", function() {
    return Tweet.deployed().then(function(instance) {
        tweetInstance = instance;
        return tweetInstance.seguir("0xc383dfb5fc71ff1bb2bbadb812229681fb
7a8e3c", { from: accounts[1]});
    }).then(function() {
        return tweetInstance.getNumSeguidos({ from: accounts[1]});
    }).then(function(num) {
        assert.equal(num, 1);
        return tweetInstance.getSeguidos(1,{ from: accounts[1]});
    }).then(function(contenido) {
        assert.equal(contenido.toUpperCase(), "0xc383dfb5fc71ff1bb2bbadb8
12229681fb7a8e3c".toUpperCase());
    });
  });
});
```

Respecto al diseño del test, este está conformado por dos pruebas, *TestAddTweet* y *TestSeguir*. Para hacer una prueba, hay que hacer uso de la istrución *it()*. Y dentro de la función, hay que usar la instrucción assert. Para pasar una prueba, es necesario que todos sus asserts se cumplan. Un assert es una instrucción que evalúa una condición. Para llamar a los diferentes métodos hacemos uso de la instrucción then, pero todo esto lo veremos en la parte del Frontend.

Respecto a al diseño de las pruebas, vamos a empezar por el primero de ellos *TestAddTweet*. Llamamos al método *addTweet()*, con el tweet *"Mensaje de prueba"*. Posteriormente llamamos al método *getNumberTweets()*, este nos debería devolver uno, el que hemos introducido previamente. Lo verificamos con

un assert. Finalmente llamamos a *getOwnTweets()* y este nos debería devolver el mensaje original "Mensaje de prueba", lo comprobamos con un assert.

La segunda prueba es *TestSeguir*. Para ello ejecutamos el método seguir y le pasamos una cuenta cualquiera. Posteriormente llamamos a *getNumberSeguidos()* y con un assert comprobamos que es uno. Finalmente llamamos al método *getSeguidos()* y con un assert comprobamos que la cuenta obtenida, es la original.

4.1.3 Frontend

Ahora vamos a hablar un poco de la parte del cliente, del frontend. Como he dicho con anterioridad este está codificado como una página web. Así que este está diseñado con HTML, CSS y JavaScript. La interacción con el contrato inteligente se realiza mediante javascript.

Vamos a ver ahora cómo interactúa nuestro cliente con el contrato, para ello primero hay que inicializar los componentes.

```
contracts: {}
web3Provider = new Web3.providers.HttpProvider('http://localhost:7545');
$.getJSON("Tweet.json", function (value) {
    contracts.Tweet = TruffleContract(value);
    contracts.Tweet.setProvider(web3Provider);
});
```

En este código podemos ver la cadena de conexión al servidor, en este caso localhost, ya que la blockchain es local. Ahora tenemos una instancia de nuestro contrato operativa con la cual podemos hacer uso de sus funciones. Para poder hacer uso de las funciones de nuestro contrato, lo haríamos de la siguiente manera. Voy a mostrar cómo sería usar la función de publicar un Tweet.

```
contracts.Tweet.deployed().then(function (instance) {
    return instance.addTweet(cadena_de_texto_a_añadir);
}).then(function (result) {
    // Código a ejecutar cuando se realice la función del contrato
})
```

Este proceso se lleva a cabo de manera asíncrona mediante el uso de promesas de JavaScript. Que de manera resumida se podría decir que, si la promesa se llega a cumplir, en este caso el añadir el "tweet", entonces se ejecuta la función en *then* en la que puedes ejecutar código en JavaScript.

De esta manera se puede llevar a cabo cada llamada a las funciones que está en nuestro contrato que hemos visto anteriormente. Al final la página queda como se observa en la Figura 13

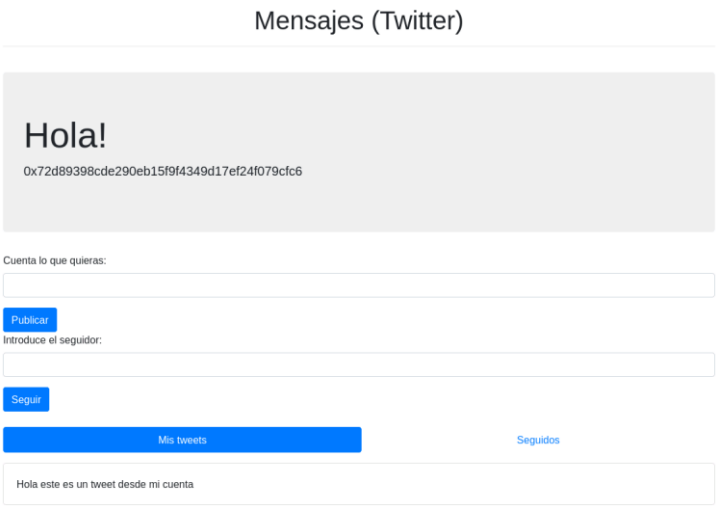


Figura 13: Pagina de Mis Tweets de "Twiter"

Aquí se puede ver un poco lo que vería el usuario, la interfaz visual. Es simple pero funcional. Un input para escribir el "tweet" y otro para introducir el código de la cuenta de usuario a seguir. Posteriormente se puede ver los "tweets" de tu cuenta y la de las personas seguidas.

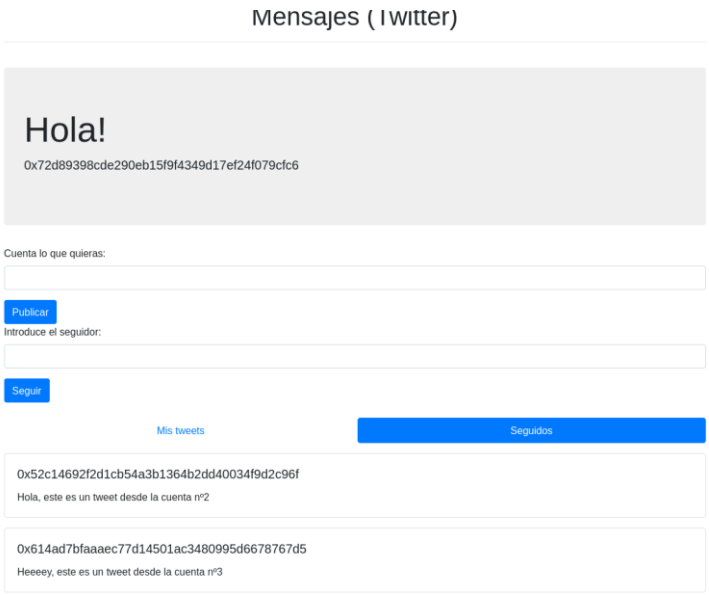


Figura 14: Pagina de Seguidos de "Twiter"

En esta última imagen, la Figura 14, estamos viendo los "tweets" de la gente a la que se sigue, así como el código de la cuenta de esa persona. Finalmente, para acabar con esta aplicación, voy a destacar una cosa. Si tenemos en cuenta que esta aplicación estuviera colgada dentro de la red principal de Ethereum y esta

estuviera funcionando con una gran cantidad de usuarios, supondría un peso en la red y este peso tiene que ser pagado. Lo que quiero decir con esto es que la persona que ha publicado el contrato ha pagado por la publicación de este, pero no de la cantidad de datos que puede albergar. Por ello cuando una persona va a guardar algo dentro de la red de blockchain, esta debe pagar por ello, en este caso en forma de ether.

En nuestro caso, si una persona quisiera publicar un tweet, debe pagar por él, al igual que si quiere seguir a una persona. Ya que estos datos tienen que ser almacenados dentro de la blockchain. Este sería uno de los aspectos en los que hay que ceder si se quiere aprovechar la parte de que las empresas no jueguen con tu privacidad. En la Figura 15 se puede ver la ventana para confirmar las transacciones por usar la aplicación.

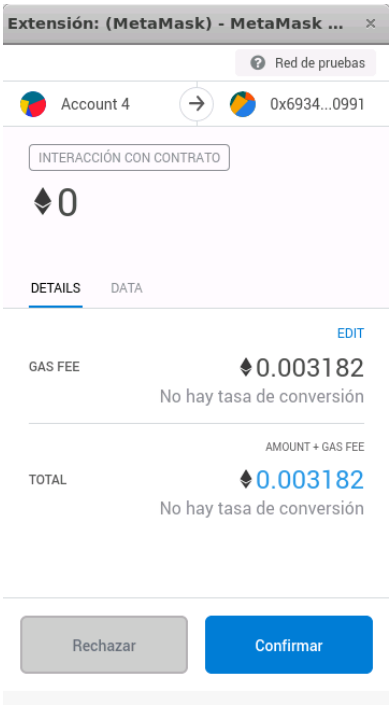


Figura 15: Pop-up que se lanza cuando se quiere publicar un "tweet"

4.2 Segundo caso de uso: Cuaderno de bitácora

4.2.1 Backend

Para llevar a cabo esta aplicación, vamos a realizar los pasos que hemos mencionado en el caso de uso anterior para tener una plantilla del proyecto que vamos a utilizar. Una vez que la tenemos bajada, vamos a realizar el contrato que llevará acabo esta funcionalidad del diario.

```
contract Diario{
```

```
    struct Entrada {
        string contenido;
        uint256 fecha;
    }

    mapping (address => mapping(uint => Entrada)) private entradas;
    mapping (address => uint) private numEntradas;

    function getOwnNumEntradas() public view returns (uint) {
        return numEntradas[msg.sender];
    }

    function setEntrada(string memory contenido, uint256 fecha) public {
        uint numEnt = getOwnNumEntradas();
        if (numEnt != 0) {
            require(entradas[msg.sender][numEnt].fecha < fecha, "La fecha introducida
 debe ser mayor que la de la anterior entrada");
        }
        numEntradas[msg.sender]++;
        numEnt = getOwnNumEntradas();
        Entrada memory nuevaEntrada = Entrada(contenido, fecha);
        entradas[msg.sender][numEnt] = nuevaEntrada;
    }

    function getContenidoEntada(uint i) public view returns (string memory) {
        return entradas[msg.sender][i].contenido;
    }

    function getFechaEntrada(uint i) public view returns (uint256) {
        return entradas[msg.sender][i].fecha;
    }

}
```

Este sería el contrato que he desarrollado para llevar a cabo el diario. Primero, para organizar el diario he pensado en crear entradas. Una entrada es un fragmento de texto acompañado de su fecha de publicación. Para poder modelar esta idea, he utilizado las estructuras que nos proporciona Solidity. Para ello hemos utilizado la palabra reservada *struct* y dentro de ella hay dos variables una es el contenido en texto de la entrada y otra es la fecha de publicación. Solidity no proporciona un tipo de datos para modelar fechas, así que vamos a guardar las fechas como enteros dentro de Solidity.

Posteriormente tenemos dos mappings. El primero asocia a cada código de cuenta de usuario una lista de entradas. Como nos pasaba en el caso de uso

anterior, no es posible realizar un array de estructuras, Solidity no lo permite. Por eso, he vuelto a utilizar un doble mapping. El segundo, es un mapping que asocia a cada usuario con el número de entradas. Una vez más, esto es necesario guardarlo porque necesitamos saber el número de entradas escritas en el primer mapping ya que no es posible saberlo de otro modo.

Ahora vamos a ver las funciones del contrato. La primera devuelve el número de entradas que ha escrito un usuario. Para ello devuelve del segundo mapping con el código de usuario de la persona que ha llamado al contrato el número de entradas.

La siguiente función nos permite escribir una entrada dentro de nuestro contrato. Para ello recibe como parámetros el contenido de la entrada y la fecha de publicación. Dentro de esta función podemos observar el nombre de otra función que corresponde a las palabras reservadas de Solidity. Esta es *require()* esta función recibe una condición, la cual se debe cumplir para poder ejecutar la función. También recibe un string que es el mensaje que se da cuando se dispara esta función. En nuestro contrato verificamos que la fecha introducida para la entrada siguiente tiene que ser superior a la de la anterior entrada. Este *require(),* aparece en rojo cuando te aparece la ventana para efectuar la transacción como vimos en el anterior caso de uso.

Posteriormente añadimos una unidad al mapping que cuenta el número de entradas, creamos una nueva variable de la estructura, la inicializamos con los parámetros de la función y la añadimos al primer mapping con el código de la cuenta del usuario que ha llamado a la función.

Se puede observar que en la declaración del primer mapping hay una palabra reservada y esta es *private*. Al principio pensaba que el contenido dentro de la blockchain estaba cifrado per se, pero no es así del todo. Lo máximo que permite Solidity de privacidad respecto al contenido de tus contratos es usar la palabra *private*, supuestamente solo permite la visibilidad a esta variable al propio contrato que la creo. Pero dentro de la máquina virtual de Ethereum, que gestiona los contratos en la red, esto parece no estar claro del todo. Así que, partiendo de la premisa de que no está del todo segura la información, vamos a aplicar otra capa por encima. Esto lo comentaremos en la parte del cliente de la aplicación.

Finalmente tenemos las dos últimas funciones del contrato. Estas dos funciones podrían ser la misma. El objetivo es devolver el contenido de la variable mapping donde se guardan las entradas. Como la devolución de estructuras no está

implementada aún, ha sido necesario realizar estas dos funciones. En una devuelve el contenido en texto de la entrada y en la otra devuelve la fecha de publicación.

4.2.2 Testing

El archivo de pruebas desarrollado para este contrato es el siguiente.

```
var Diario = artifacts.require("./Diario.sol");

contract("Diario", function(accounts) {
  var DiarioInstance;

  it("TestSetEntrada", function() {
    return Diario.deployed().then(function(instance) {
        DiarioInstance = instance;
        return DiarioInstance.setEntrada("Contenido", 00000, { from: acco
unts[1] });
    }).then(function() {
        return DiarioInstance.getOwnNumEntradas({ from: accounts[1] });
    }).then(function(num) {
        assert.equal(num, 1);
    });
  });

  it("TestGetEntrada", function() {
    return Diario.deployed().then(function(instance) {
        DiarioInstance = instance;
        return DiarioInstance.getOwnNumEntradas({ from: accounts[1] });
    }).then(function(num) {
        assert.equal(num, 1);
        return DiarioInstance.getContenidoEntada(1, { from: accounts[1] }
);
    }).then(function(contenido) {
        assert.equal(contenido, "Contenido");
        return DiarioInstance.getFechaEntrada(1, { from: accounts[1] });
    }).then(function(fecha) {
        assert.equal(fecha, 00000);
    });
  });
});
```

En este test, tenemos dos pruebas al igual que el anterior. La primera de ella es *TestSetEntrada*. En esta prueba comprobamos que se introduce una entrada en el diario. Primero llamamos al método setEntrada() con una cadena de texto: "Contenido" y un entero, que representa la fecha: 00000. Después llamamos al

método *getOwnNumEntradas()* y verificamos con un assert que el número de entradas es uno.

La segunda es *TestGetEntrada*. En esta prueba verificamos que el contenido introducido en la anterior prueba es el que hemos introducido. Me gustaría resaltar que entre prueba y prueba no se reinicia el contrato, con lo que una prueba puede afectar a otra, como es este caso. Nos valemos de lo introducido previamente para verificarlo.

Comenzamos repitiendo la llamada *getOwnNumEntradas()* y con un assert comprobamos que sea una. Posteriormente hacemos la llamada al método *getContenidoEntrada()* y comprobamos que la cadena de caracteres devuelta sea "Contenido". Finalmente, llamamos al método *getFechaEntrada()* y comprobamos que el valor devuelto sea 00000.

4.2.3 Frontend

Ahora vamos a comentar la parte del cliente. He añadido una librería nueva, esta se trata de *CriptoJs.* Esta librería implementa un montón de algoritmos de cifrado conocidos. En nuestro caso vamos a utilizar el algoritmo de cifrado por bloques de clave simétrica AES. Para poder usarlo en nuestro código vamos a poner la siguiente sentencia en el archivo HTML.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-
js/3.1.2/rollups/aes.js"></script>
```

Como hemos visto en el caso de uso anterior, inicializaremos nuestra instancia del contrato y una vez la tengamos ya podremos hacer llamadas al contrato dentro de la blockchain siguiendo la estructura de las promesas de JavaScript.

Dentro del código hay dos funciones, la de escribir y la de mostrar el texto. Las dos requieren la contraseña para poder cifrar y descifrar el contenido respectivamente. A continuación, voy a insertar un fragmento del código de cómo sería.

```
var fecha = Number((new Date()).getTime());
App.contracts.Diario.deployed().then(function (instance) {
      var encrypted = CryptoJS.AES.encrypt(contenido, pass_str);
      encrypted = String(encrypted);
      return instance.setEntrada(encrypted, fecha);
    }).then(function (result) {
      alert("Se ha guardado correctamente");
      location.reload();
```

```
    })
```

En este fragmento podemos ver como obtenemos la fecha. Primero obtenemos la fecha con la variable Date de JavaScript, posteriormente la convertimos en número. Esta nos devuelve los segundos que han pasado desde el 1 de enero de 1970. Con esto ya lo tenemos en formato de entero para poder guardarlos en el contrato. Después se puede ver como aplicamos la función de AES para cifrar el contenido. Finalmente lo introducimos en el contrato y notificamos si todo ha ido bien.

La función de leer las entradas es similar y la muestro a continuación.

```
App.contracts.Diario.deployed().then(function (instance) {
      diarioInstance = instance;
      return instance.getOwnNumEntradas();
    }).then(function (numEntradas) {
      $('#getEntradas').hide();
      if (numEntradas == 0) {
        // notificamos que no hay entradas
        return;
      }
      for (let i = 1; i <= numEntradas; i++) {
        diarioInstance.getContenidoEntada(i).then(function (contenido){
          diarioInstance.getFechaEntrada(i).then(function (fechaNum){
            var decrypted = CryptoJS.AES.decrypt(contenido, pass_str).toString(Crypto
JS.enc.Utf8);
            fechaNum = Number(fechaNum);
            var fecha = new Date(fechaNum);
            // Escribimos la nueva entrada en la página
          })
        });
      }
    })
```

En este caso primero obtenemos el número de entradas para así poder recorrer el mapping del contrato. Si no hay lo notificamos en la página. Cuando obtenemos la entrada para descifrar el contenido de texto, volvemos a llamar a la función de AES con la contraseña. Hacer mención también a la llamada al método *toString()* con la codificación UTF8 para que el string devuelto cumpla con los caracteres correctos. Finalmente, transformamos el entero que representa la fecha en el tipo Date de Javascript y montamos la entrada en la página.

El resultado final de la página quedaría tal que así.

Figura 16: Página web del Diario

En la Figura 16 podemos ver que tenemos las dos funcionalidades antes mencionadas en la página web. Primero tenemos un input en el que poner la contraseña y al darle a descifrar trae las entradas, las descifra y las muestra. Después tenemos un área de texto para escribir el contenido de nuestra entrada y un input para poner la contraseña. Al darle al botón de guardar, esta se cifrará y se guardará en la blockchain. En la figura 17 podemos ver cómo se muestran las entradas escritas por un usuario.

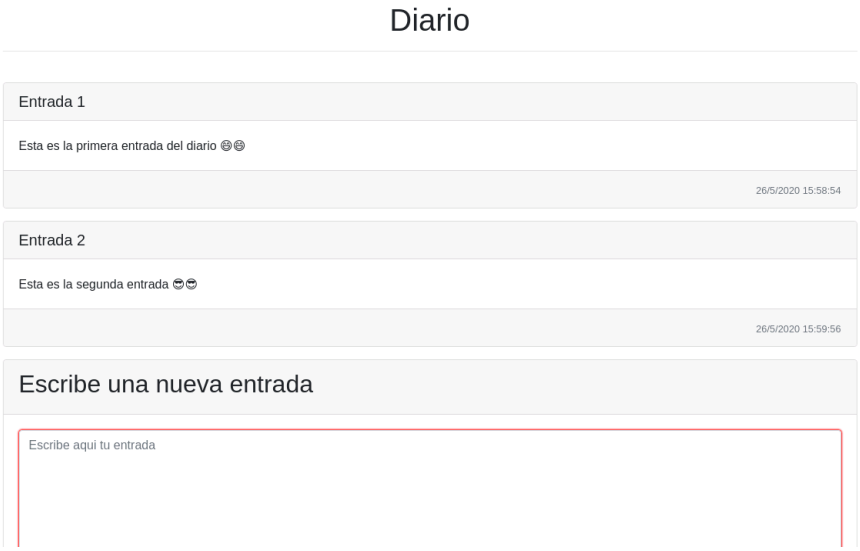


Figura 17: Entradas de la página del Diario

Volver a recordar una vez más que cada vez que queramos guardar algo dentro de la blockchain a través del contrato, esto no será gratuito. Cuando escribamos una nueva entrada en el diario esta nos costará ether.

4.3 Tercer caso de uso: Tres en raya

4.3.1 Backend

A continuación, muestro el contrato que gestiona el juego de tres en raya.

```
contract TresRaya {
    enum Opciones {vacio, jugador1, jugador2}

    address player1;
    address player2;
    Opciones turno;
    Opciones[3][3] tablero;

    function reset() public {
        player1 = address(0);
        player2 = address(0);
        turno = Opciones.vacio;
        tablero[0][0] = Opciones.vacio;
        tablero[0][1] = Opciones.vacio;
        tablero[0][2] = Opciones.vacio;
        tablero[1][0] = Opciones.vacio;
        tablero[1][1] = Opciones.vacio;
        tablero[1][2] = Opciones.vacio;
        tablero[2][0] = Opciones.vacio;
        tablero[2][1] = Opciones.vacio;
        tablero[2][2] = Opciones.vacio;
    }

    constructor() public {
        reset();
    }

    function hayHueco() public view returns (bool) {
        return player1 == address(0) || player2 == address(0);
    }

    function hayJugador1() public view returns (bool) {
        return player1 != address(0);
    }

    function setPlayer1() public {
        player1 = msg.sender;
    }

    function setPlayer2() public {
        player2 = msg.sender;
    }

    event movimientoRealizado(
```

```
        uint x,
        uint y,
        uint turno,
        uint ganador
    );

    function movimiento(uint x, uint y) public {
        if (turno == Opciones.vacio) {
            turno = Opciones.jugador1;
        }
        require(tablero[x][y] == Opciones.vacio, "Posición no valida");
        tablero[x][y] = turno;
        Opciones ganador = hayGanador();
        uint intganador;
        uint intturno;
        if (ganador == Opciones.jugador1) {
            intganador = 1;
        }
        if (ganador == Opciones.jugador2) {
            intganador = 2;
        }
        if (ganador == Opciones.vacio) {
            intganador = 0;
        }
        if(hayEmpate()){
            intganador = 3;
        }
        if (turno == Opciones.jugador1) {
            intturno = 1;
        }
        if (turno == Opciones.jugador2) {
            intturno = 2;
        }
        emit movimientoRealizado(x, y, intturno, intganador);
        if (ganador != Opciones.vacio || intganador == 3) {
            reset();
        } else {
            if (turno == Opciones.jugador1) {
                turno = Opciones.jugador2;
            } else {
                turno = Opciones.jugador1;
            }
        }
    }

    function hayEmpate() private view returns (bool) {
        bool salida = true;
        for (uint i = 0; i < 3; i++) {
            for (uint j = 0; j < 3; i++) {
```

```
                if(tablero[i][j] == Opciones.vacio){
                    salida = false;
                    return salida;
                }
            }
        }
        return salida;
    }

    function hayGanador() private view returns (Opciones) {
        // Comprobamos las filas y las columnas
        for (uint i = 0; i < 3; i++) {
            // filas
            if (
                tablero[i][0] == tablero[i][1] && tablero[i][0] == tablero[i][2]
            ) {
                if (tablero[i][0] == Opciones.jugador1) {
                    return Opciones.jugador1;
                }
                if (tablero[i][0] == Opciones.jugador2) {
                    return Opciones.jugador2;
                }
            }
            // columnas
            if (
                tablero[0][i] == tablero[1][i] && tablero[0][i] == tablero[2][i]
            ) {
                if (tablero[0][i] == Opciones.jugador1) {
                    return Opciones.jugador1;
                }
                if (tablero[0][i] == Opciones.jugador2) {
                    return Opciones.jugador2;
                }
            }
        }
        // Diagonales
        if (tablero[0][0] == tablero[1][1] && tablero[0][0] == tablero[2][2]) {
            if (tablero[0][0] == Opciones.jugador1) {
                return Opciones.jugador1;
            }
            if (tablero[0][0] == Opciones.jugador2) {
                return Opciones.jugador2;
            }
        }
        if (tablero[0][2] == tablero[1][1] && tablero[0][2] == tablero[2][0]) {
            if (tablero[0][2] == Opciones.jugador1) {
                return Opciones.jugador1;
            }
            if (tablero[0][2] == Opciones.jugador2) {
```

```
                return Opciones.jugador2;
            }
        }
        // Si no se ha encontrado ganador devolvemos vacio
        return Opciones.vacio;
    }
}
```

Como se puede observar, este es un poco más extenso que en los casos anteriores, también es normal debido a que maneja más cosas. Comencemos a comentarlo.

Al principio nos encontramos con una sentencia nueva, esta se trata de un enum, una enumeración. Esta se llama *Opciones* en ella se recogen tres valores: vacío, jugador1 y jugador2. Esto lo usaremos después para construir otros datos. Después tenemos dos variables de tipo *address* en ellas guardaremos los usuarios que serán el jugador 1 y el jugador 2.

Después tenemos una variable que recoge de quien es turno, si es del jugador 1, o del jugador 2. Esta es del tipo enum mencionado anteriormente. Finalmente tenemos el tablero de juego, este es un doble array de tipo *Opciones*, otra vez el enum. En otras palabras, este será el tablero que represente la partida y cada celda que lo compone puede ser o del jugador 1, el jugador 2 o vacía.

Ahora hablaremos de las funciones. Primero nos encontramos con la función reset, esta será la encargada de volver todas las variables a sus valores por defecto. Las variables que guardan a los jugadores, las pone a la cuenta nula que es *address(0)*. Establece el turno a vacío, pues no hay una partida en curso. Finalmente inicializa cada celda del tablero a valor vacío.

A continuación, tenemos una función un tanto especial, que aún no habíamos visto. Esta se trata del constructor del contrato. A esta función se le llama la primera vez que se llama al contrato y permite inicializarlo. Anteriormente no nos había hecho falta usarla. En nuestro caso lo que hace es llamar a la función *reset()* para inicializar todas las variables.

En la siguiente función nos permite saber si ya se han asignado dos jugadores y por lo tanto la partida está llena. La función *hayJugador1(),* nos devuelve si el jugador uno ha sido establecido o no. A continuación, tenemos dos funciones para establecer el jugador uno y el jugador 2 respectivamente.

La siguiente sentencia que se puede observar, también es una sentencia nueva, esta se trata de un evento. Los eventos son un tipo de datos que se envían al cliente de manera asíncrona. Esto nos permitirá, como veremos más adelante, poder modificar el contenido de la página sin tener que recargar la misma. En el evento declaramos el número de parámetros que queremos pasar. En este caso tenemos cuatro enteros. Los enteros representan las dos coordenadas, de quién es el turno y si hay ganador o empate;

Después tenemos una de las funciones más importante que controla la lógica del juego. Esta es *movimiento(x, y).* Esta recibe como parámetros, las coordenadas del tablero donde se produce la jugada. Primero verificamos que el turno esté vacío. Si es así, es el turno del jugador 1. Después tenemos una sentencia que hemos visto en el anterior caso de uso, *require()*. En ella verificamos que la posición que se ha indicado es de tipo vacío.

Después le asignamos a la posición en la tabla el valor del valor del turno. A continuación, verificamos si hay ganador o empate con las funciones que veremos tras acabar con esta. Seguidamente enviamos el evento que previamente hemos creado con los datos que he mencionado antes. Para lanzar el evento se usa la palabra reservada *emit*. Posteriormente si la partida se ha acabado reseteamos el contrato con *reset()*. Si la partida aún no se ha acabado, cambiamos el turno. Si es del jugador1 se cambia al jugador 2 y viceversa.

Finalmente nos encontramos con las funciones para verificar si hay un empate y si hay un ganador. La función que verifica el empate verifica que todas las celdas no están vacías y la función que verifica si hay ganador verifica por cada jugador si hay tres celdas consecutivas con el mismo valor.

4.3.2 Testing

A continuación, muestro el test para el contrato que implementa el tres en raya.

```
var TresRaya = artifacts.require("./TresRaya.sol");

contract("TresRaya", function(accounts) {
  var TresRayaInstance;

  it("TestReset", function() {
    return TresRaya.deployed().then(function(instance) {
        TresRayaInstance = instance;
        return TresRayaInstance.reset();
    }).then(function() {
        return TresRayaInstance.hayHueco();
    }).then(function(result) {
```

```
        assert.equal(result, true);
        return TresRayaInstance.hayJugador1();
    }).then(function(result) {
        assert.equal(result, false);
    });
});

it("TestSimularGana1", function() {
    return TresRaya.deployed().then(function(instance) {
        TresRayaInstance = instance;
        return TresRayaInstance.setPlayer1({ from: accounts[1] });
    }).then(function() {
        return TresRayaInstance.hayJugador1();
    }).then(function(result) {
        assert.equal(result, true);
        return TresRayaInstance.setPlayer2({ from: accounts[2] });
    }).then(function(result) {
        return TresRayaInstance.hayHueco();
    }).then(function(result) {
        assert.equal(result, false);
        return TresRayaInstance.movimiento(0, 0);
    }).then(function() {
        return TresRayaInstance.movimiento(1, 0);
    }).then(function() {
        return TresRayaInstance.movimiento(0, 1);
    }).then(function() {
        return TresRayaInstance.movimiento(1, 1);
    }).then(function() {
        return TresRayaInstance.movimiento(0, 2);
    }).then(function() {
        return TresRayaInstance.hayHueco();
    }).then(function(result) {
        assert.equal(result, true);
    });
});

it("TestSimularGana2", function() {
    return TresRaya.deployed().then(function(instance) {
        TresRayaInstance = instance;
        return TresRayaInstance.setPlayer1({ from: accounts[1] });
    }).then(function() {
        return TresRayaInstance.hayJugador1();
    }).then(function(result) {
        assert.equal(result, true);
        return TresRayaInstance.setPlayer2({ from: accounts[2] });
    }).then(function(result) {
        return TresRayaInstance.hayHueco();
    }).then(function(result) {
        assert.equal(result, false);
```

```
        return TresRayaInstance.movimiento(0, 0);
    }).then(function() {
        return TresRayaInstance.movimiento(1, 0);
    }).then(function() {
        return TresRayaInstance.movimiento(0, 1);
    }).then(function() {
        return TresRayaInstance.movimiento(1, 1);
    }).then(function() {
        return TresRayaInstance.movimiento(2, 2);
    }).then(function() {
        return TresRayaInstance.movimiento(1, 2);
    }).then(function() {
        return TresRayaInstance.hayHueco();
    }).then(function(result) {
        assert.equal(result, true);
    });
  });

  it("TestSimularEmpate", function() {
    return TresRaya.deployed().then(function(instance) {
        TresRayaInstance = instance;
        return TresRayaInstance.setPlayer1({ from: accounts[1] });
    }).then(function() {
        return TresRayaInstance.hayJugador1();
    }).then(function(result) {
        assert.equal(result, true);
        return TresRayaInstance.setPlayer2({ from: accounts[2] });
    }).then(function(result) {
        return TresRayaInstance.hayHueco();
    }).then(function(result) {
        assert.equal(result, false);
        return TresRayaInstance.movimiento(0, 0);
    }).then(function() {
        return TresRayaInstance.movimiento(1, 0);
    }).then(function() {
        return TresRayaInstance.movimiento(0, 1);
    }).then(function() {
        return TresRayaInstance.movimiento(0, 2);
    }).then(function() {
        return TresRayaInstance.movimiento(1, 1);
    }).then(function() {
        return TresRayaInstance.movimiento(2, 2);
    }).then(function() {
        return TresRayaInstance.movimiento(1, 2);
    }).then(function() {
        return TresRayaInstance.movimiento(2, 0);
    }).then(function() {
        return TresRayaInstance.movimiento(2, 1);
    }).then(function() {
```

```
        return TresRayaInstance.hayHueco();
    }).then(function(result) {
        assert.equal(result, true);
    });
  });
});
```

En este test tenemos cuatro pruebas. Estas son: *TestReset*, *TestSimularGana1*, *TestSimularGana2* y *TestSimularEmpate*. La primera verifica que se pueda reiniciar la partida y las demás simulan una partita verificando las tres posibles variantes que se pueden dar en una partida.

Una cosa a tener en cuenta es que en este contrato algunos de los métodos se encuentran en privado, con lo que esos no se pueden probar. Tampoco se pueden usar dentro de una prueba para probar otros métodos. Dicho esto, las pruebas desarrolladas abarcan todos los escenarios posibles.

En la primera función TestReset realizamos lo siguiente. Efectuamos el método reset. Posteriormente llamamos al método hayHueco(), con un assert comprobamos que el valor devuelto sea verdadero. Finalmente llamamos al método hayJugador1() y comprobamos con un assert que el valor devuelto sea falso.

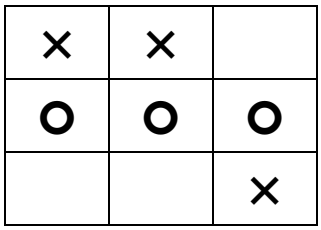
La siguiente prueba es *TestSimularGana1()*. Primero llamamos al método *setPlayer1()*, esto lo hacemos con la primera cuenta que se encuentra en Ganache. Luego llamamos al método *hayJugador1()* y con un assert comprobamos que es cierto. Posteriormente llamamos al método setPlayer2() con la segunda cuenta que se encuentra en Ganache y comprobamos con esto con un assert y el método hayhueco() que devuelva el valor falso.

A continuación, vamos a simular una partida. Para ello se ejecuta el método moviemento(). Este método lo vamos a ejecutar varias veces, este caso simulando que gana el jugador 1. Siendo ✕ el jugador 1 y ○ el jugador 2, la partida que se va a llevar a cabo va a ser la siguiente.

✕	✕	✕
○	○	

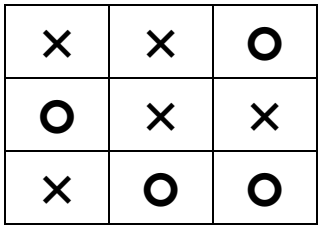
Finalmente, como no se puede comprobar quien es el ganador porque es una función privada, lo que vamos a hacer es suponer que eso ha ocurrido y que el contrato se ha reseteado. Para ello hacemos una llamada al método hay hueco y que este devuelva el valor verdadero. Comprobamos con un assert que esto sea así.

La prueba que está a continuación es *TestSimularGana2()*. Esta prueba es similar a la anterior, pero cuando llegue el momento de realizar la llamada a los métodos de *movimiento()*, simularemos que gana el jugador 2. Siendo ✕ el jugador 1 y ○ el jugador 2, la partida que se va a llevar a cabo va a ser la siguiente.



Finalmente comprobamos que *hayHueco()* devuelva verdadero.

La última prueba es *TestSimularEmpate*. Similar a las anteriores, realizamos el método *movimiento()* y simulamos la partida. Siendo ✕ el jugador 1 y ○ el jugador 2, la partida que se va a llevar a cabo va a ser la siguiente.



Finalmente, una vez más llamamos al método *hayHueco()* y comprobamos con un assert que este devuelva verdadero.

4.3.3 Frontend

Una vez más necesitamos las librerías para el manejo del contrato que hemos mencionado en el primer caso de uso. También he vuelto a utilizar el framework de Bootstrap. Como siempre, primero hay que inicializar el contrato, pero esta vez pondremos una sentencia de más. Esta nos permitirá escuchar los eventos que se emitan.

```
App.listenForEvents();
```

Primero vamos a comentar la primera función que va a ser la de buscar partida, su código es el siguiente.

```
App.contracts.TresRaya.deployed().then(function (instance) {
      tresRaya = instance;
      return tresRaya.hayHueco();
    }).then(function (result) {
      if (result) {
        tresRaya.hayJugador1().then(function (result2) {
          if (!result2) {
            tresRaya.setPlayer1().then(function () {
              // se ha establecido el jugador 1
            });
          } else {
            tresRaya.setPlayer2().then(function () {
              // Se ha establecido el jugador 2
            });
          }
        });
      } else {
        alert("Ya hay una partida en curso");
      }
    })
```

Primero verificamos si hay hueco en la partida, si no lo hay lo notificamos. Si hay hueco, verificamos si hay un jugador 1. Si lo hay establecemos ese cliente como el jugador 2 y si no lo hay lo establecemos como el jugador 1. Con esto ya tendríamos establecida una partida.

Para seleccionar una posición para realizar la jugada he realizado una tabla en HTML de 3 por 3, que veremos a continuación. Por cada celda de la tabla he creado la siguiente función que se ejecuta al hacer clic en la celda.

```
function click1() {
  App.contracts.TresRaya.deployed().then(function (instance) {
    return instance.movimiento(2, 0);
  }).then(function (result) {
    // console.log(result);
  }).catch(function (err) {
    console.error(err);
  });
}
```

Esta función *clickN()* de JavaScript, ejecuta la función *movimiento()* de nuestro contrato. Dependiendo de la celda pulsada se ejecutará la función *clickN()* siendo

este N el número de la casilla el cual ha pulsado el jugador. Cada función llamara a *movimiento()* pasándole como parámetros las coordenadas de la celda pulsada.

En el siguiente fragmento de código se muestra cómo se recibe el evento creado en el contrato. Como hemos visto en el contrato, este evento nos devuelve las coordenadas de la posición seleccionada, de quien es el turno y por lo tanto que forma poner en el tablero y finalmente si ha habido un ganador, empate o se sigue jugando.

```
listenForEvents: function () {
    var instancia;
    App.contracts.TresRaya.deployed().then(function (instance) {
      instancia = instance;
      instance.movimientoRealizado({}, {
        fromBlock: 0,
        toBlock: 'latest'
      }).watch(function (error, event) {
        var x = Number(event.args.x);
        var y = Number(event.args.y);
        var turno = Number(event.args.turno);
        var ganador = Number(event.args.ganador);
        if (turno == 1) {
          if (x == 0 && y == 0) { $("#td7").text(App.marcador1); }
          if (x == 0 && y == 1) { $("#td8").text(App.marcador1); }
          if (x == 0 && y == 2) { $("#td9").text(App.marcador1); }
          if (x == 1 && y == 0) { $("#td4").text(App.marcador1); }
          if (x == 1 && y == 1) { $("#td5").text(App.marcador1); }
          if (x == 1 && y == 2) { $("#td6").text(App.marcador1); }
          if (x == 2 && y == 0) { $("#td1").text(App.marcador1); }
          if (x == 2 && y == 1) { $("#td2").text(App.marcador1); }
          if (x == 2 && y == 2) { $("#td3").text(App.marcador1); }
          $("#informacion").text("Es el turno del jugador 2");
        } else {
          if (x == 0 && y == 0) { $("#td7").text(App.marcador2); }
          if (x == 0 && y == 1) { $("#td8").text(App.marcador2); }
          if (x == 0 && y == 2) { $("#td9").text(App.marcador2); }
          if (x == 1 && y == 0) { $("#td4").text(App.marcador2); }
          if (x == 1 && y == 1) { $("#td5").text(App.marcador2); }
          if (x == 1 && y == 2) { $("#td6").text(App.marcador2); }
          if (x == 2 && y == 0) { $("#td1").text(App.marcador2); }
          if (x == 2 && y == 1) { $("#td2").text(App.marcador2); }
          if (x == 2 && y == 2) { $("#td3").text(App.marcador2); }
          $("#informacion").text("Es el turno del jugador 1");
        }
        if (ganador != 0) {
          if (ganador == 1) {
            alert("Ha ganado el Jugador 1");
```

```
        } else if(ganador == 2) {
          alert("Ha ganado el jugador 2");
        } else {
          alert("Empate");
        }
        location.reload();
      }
    });
  });
},
```

Esta función se ejecuta cada vez que le llega un evento. Primero con las coordenadas y el turno que hemos recibido, dibujamos en pantalla ❌ o ⭕ dependiendo del turno del jugador. Jugador 1 será ❌ y jugador 2 será ⭕. Posteriormente con el dato de quien ha ganado o si ha habido un empate, lo notificamos al usuario y recargamos la página. Recordemos, que llegados a este punto el contrato se ha reseteado.

Ahora vamos a ver cómo ha quedado la página.

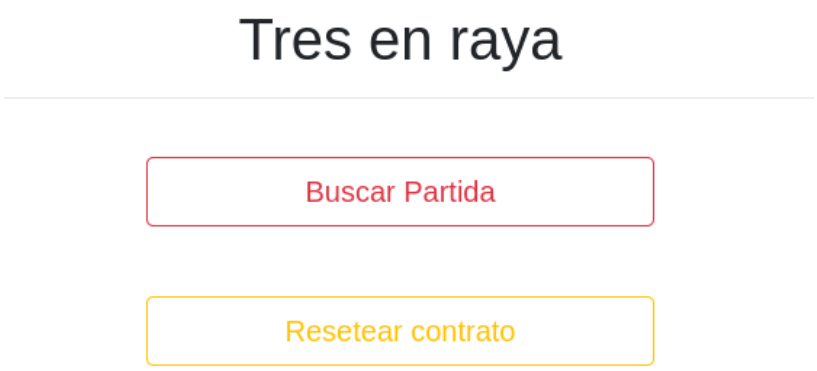


Figura 18: Pantalla de buscar partida

En la Figura 18 tenemos la pantalla de buscar partida que ejecuta el código que hemos dicho anteriormente. El botón de resetear contrato lo mencionaremos más adelante.

Figura 19: Partida en curso del tres en raya

En la Figura 19 podemos ver una partida en curso. Como he mencionado anteriormente para realizar una jugada basta con pulsar en la celda deseada. También hay que destacar que hay dos marcadores encima del tablero en el cual identificamos quien es el jugador 1 y quien es el jugador 2 y también mostramos de quién es el turno.

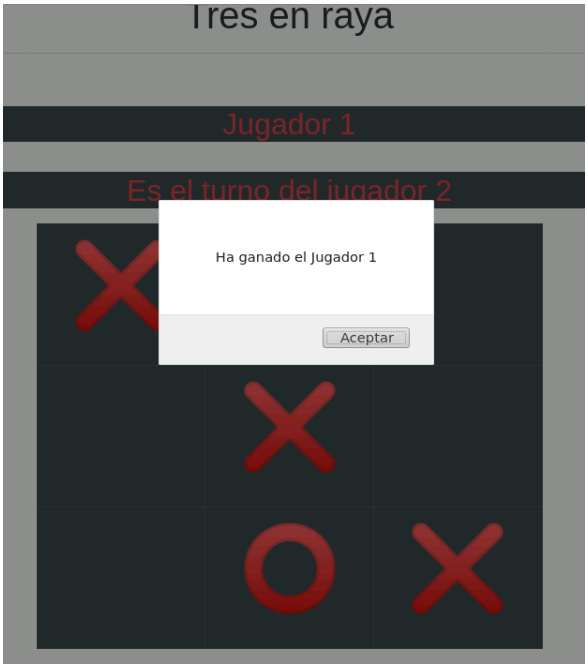


Figura 20: Final de la partida de tres en raya

En esta última imagen, la Figura 20, podemos ver el mensaje final que aparece cuando se termina la partida. Acto seguido la página se recargará volviendo al menú de buscar partida.

5 Análisis de los casos de uso

En este apartado vamos a analizar cada uno de los casos de uso. Vamos a comparar como ha sido el resultado final obtenido frente al esperado, como podría ser el de una tecnología convencional.

Empezaremos hablando del despliegue de los contratos. Cuando desplegamos un contrato dentro de la red obtenemos el resultado que se puede ver en la Figura 21.

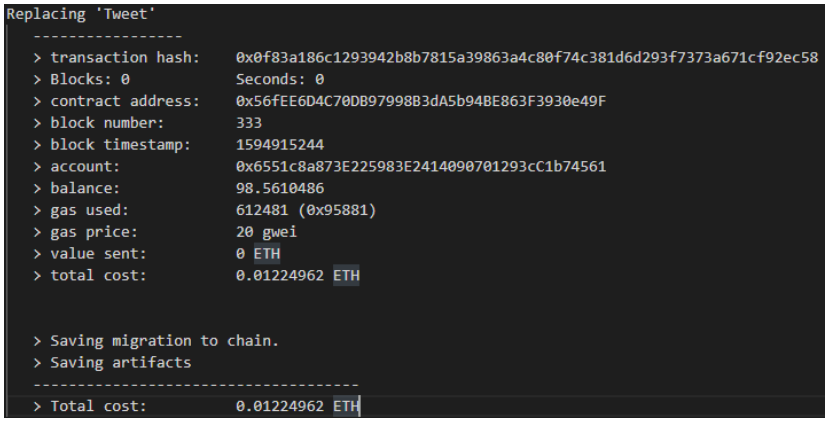


Figura 21 Despliegue de la red social

Como se puede observar, este es el caso de uso de la red del coste de desplegar este contrato en una red de Ethereum son 0.01224962 ETH que al cambio en euros actualmente es de 2.49 Euros. Recordemos que nos encontramos en una red de desarrollo y este coste no es real.

Una vez el contrato es desplegado este permanece en la red, con lo que este sería el coste total de la parte del servidor. Visto desde esta perspectiva, parece un tanto barato, ya que, de manera habitual una aplicación se encuentra en un servidor y este a su vez presenta un alquiler periódico. Con lo que pagar la cantidad antes mencionada es prácticamente ínfimo con respecto a la otra alternativa.

Comparándolo con los otros dos casos de uso, tenemos que la aplicación de diario cuesta desplegarla 0.00837924 ETH que en euros son 1.71. La aplicación de tres en raya nos cuesta 0.02015526 ETH que son 4,10 euros.

El precio varía en función de diversos factores, pero uno de ellos es la memoria que ocupa de por sí el contrato con todas las variables que no son dinámicas. Por eso, se puede apreciar cómo el contrato del tres en raya vale más. Tenemos un

array doble declarado y más funciones que en los otros contratos. En cualquier caso, sigue siendo más rentable que un despliegue convencional.

Por supuesto, no todo esto es tan bueno. Como hemos mencionado con anterioridad, esto está pensado para un uso de Smart contracts, no para aplicaciones. Por lo que cada interacción con el contrato tiene un precio. Vamos a analizar en cada caso de uso cómo afecta esto.

En el caso de la red social, nos surgen dos incidencias. Una es a la hora de publicar un tweet, este cuesta una cantidad publicarlo, en la Figura 12, podemos ver el valor de este. En este caso el precio del tweet es variable y esto depende de la cantidad de caracteres que escribamos, o lo que es lo mismo, lo que ocupa la cadena de texto dentro de la red.

Luego tenemos la funcionalidad de seguir a un usuario. En este caso, como lo que se almacena dentro del contrato al realizar la función de seguir es el código en hexadecimal que identifica al usuario. Cada vez que decidimos seguir aun usuario este tiene un coste fijo. Este es 0.001937 ETH que en euros son 0.40.

Con lo que hemos visto anteriormente, estamos viendo como esto nos abre las puertas a otra modalidad de negocio a las redes sociales. A cambio de no vender tu privacidad a una empresa, pagas por las interacciones que realices dentro de la aplicación.

Ahora veremos el caso del Diario. En este caso, por cada usuario guardamos varias entradas y estas están formadas por su fecha de publicación y por su cadena de texto. Nos encontramos ante un caso similar al anterior, como cada entrada es variable el precio por guardar cada entrada es variable.

Viendo un ejemplo práctico, si quisiéramos guardar la siguiente entrada: "Esta es la primera entrada del diario", esta nos costaría 0.004445 ETH que son 0.91 euros. Nos encontramos con otro modelo de negocio otra vez. En vez de encargar nuestros datos a una empresa, de la cual dependemos para su persistencia, tenemos que pagar por el almacenamiento de cada dato.

Finalmente tenemos el caso del juego, en este caso como se puede suponer por los anteriores por cada interacción esta nos será cobrada, ya que estamos almacenado algo dentro de la red. Como ejemplo vamos a poner cuanto sería el valor de realizar un movimiento, 0.00303 ETH que al cambio son 0.62 euros. Como se puede ver en este caso nos encontramos con un modelo que no resulta muy viable.

Ahora voy a comentar una serie de problemas y observaciones que he tenido en el desarrollo del tercer caso de uso, el del tres en raya, este ha sido el más conflictivo. Primero voy a comentar el botón de resetear contrato que hemos visto en la Figura 18. Este lo he usado mientras probaba la aplicación, este simplemente ejecuta la función *reset()* del contrato.

Uno de los problemas que me ha surgido ha sido con el tema de los valores que devuelven las funciones del contrato. Algunas funciones, como por ejemplo es la de *movimiento(),* devolvía un objecto que contenía datos de la blockchain como por ejemplo número de bloque, transacciones y demás. Pero no devolvía los valores correctos. Debido a ello he tenido que pasar más argumentos al evento de lo que tenía pensado.

Otra cosa para comentar, como hemos visto en los anteriores casos de uso, cada vez que es necesario guardar algo dentro de una variable del contrato, es necesario pagar por ello. En este caso esto nos supone una molestia, ya que por cada clic efectuado para realizar una jugada es necesario realizar una de estas transacciones. También es necesario para poder resetear el contrato.

También hay que comentar que debido a mi entorno de desarrollo y que el login de las cuentas se efectúa en Metamask. He tenido diversos problemas a la hora de probar la aplicación. Tanto si lo abro en dos pestañas diferentes o en otro navegador, de tal manera que tenga la página en dos sitios distintos. A la hora de tener que validar una transacción como puede ser entrar en la partida o hacer un movimiento, esta se repite varias veces, cuando solo afecta una de ellas. Otra cosa a comentar también es que el hecho de cambiar de cuenta dentro de Metamask no afecta al contrato, ya que solo cogerá una de ellas.

Por este motivo he tenido que quitar algunas restricciones (*require()*) que tenía previamente el contrato que era un poco evidentes. Estas eran, la comprobación de que las dos cuentas que entran en el juego deben ser diferentes y que a la hora de hacer un movimiento solo se efectué de la cuenta en la cual es su turno.

Finalmente comentar un problema que hay con los eventos. El contrato genera una cola de eventos que se van guardando y cada vez que un contrato se inicializa, recibe todos los eventos que el contrato ha enviado desde que se creó. Por lo que cuando el contrato se resetea para volver a empezar una partida, sigue saltando el mensaje de que ha habido un ganador. Esta es una característica que presenta Solidity, por lo que no se puede cambiar.

6 Conclusiones y líneas futuras

6.1 Conclusiones

Con todo lo que hemos visto, podemos hacer una pequeña valoración de lo que la tecnología Blockchain puede ofrecernos. Partimos de la base de que la tecnología de Blockchain aún es una tecnología en auge y que probablemente aún le quede mucho recorrido hasta que llegue a su cumbre.

También vamos a poder ver qué cosas nos pueden traer en un futuro esta tecnología, ya que hemos visto que no solo se ha limitado al uso de criptomonedas, sino que ha ido más allá. Tampoco hay que quitarles mérito a las criptomonedas ya que son el principal motor que sustenta la gran mayoría de redes de Blockchain.

Hemos visto cómo existe una variedad de alternativas y que cada una implementa una Blockchain con diferentes funcionalidades y que presenta una solución para el mundo empresarial. Con lo que esto potenciará sus posibles características en un futuro.

Otra de las cosas que hemos visto es que Bitcoin empezó y sigue siendo la más conocida, pero que por ejemplo Ethereum no tiene nada de que envidarle. Ha desarrollado su propio entorno de desarrollo para la creación de contratos inteligentes y además ha desarrollado su propio lenguaje de programación, Solidity.

Solidity ha sido el lenguaje que hemos utilizado para desarrollar los casos de uso. Este lenguaje me ha sorprendido, no solo presenta una solución cómoda a la hora de desarrollar contratos inteligentes, sino que además este te abstrae del manejo de la memoria de la máquina virtual de Ethereum. También me ha sorprendido en el sentido de que permite la utilización de sentencias más complejas como son los *structs* y los *enum*. Además, utilizar estas sentencias ha sido muy intuitivo, lo que ha facilitado el desarrollo de los casos de uso. También me ha gustado la interacción con JavaScript. Ya me había tocado utilizar el sistema de las promesas de JavaScript con lo cual no me ha supuesto mucha dificultad.

Ahora voy a mencionar cosas que no me han terminado de convencer de Solidity. Aunque se permitan el uso de variables un poco más complejas de lo esperado, estas no se pueden usar como devolución de una función para traerlas a JavaScript. Me estoy refiriendo sobre todo a los arrays de Strings o a las structs. Por lo que he podido leer, se prevé que en futuras versiones de Solidity esto se implemente. Otra de las cosas que no están implementadas es el tema de los

eventos. Como he comentado anteriormente desde que un contrato es publicado, todos los eventos que se lancen se mantienen. Es decir que cuando un cliente se conecte al contrato, recibirá toda la cola de eventos. Debería poderse borrar esta cola de eventos. En cualquier caso, esto parece estar implementado con el objetivo de solo notificar que pase algo, no que trasfieran datos. Como por ejemplo actualizar la vista de la página web.

Tampoco me ha gustado la idea de que parte de estas limitaciones en Solidity hagan que sea obligatorio delegar parte de la lógica de negocio al cliente. Me hubiera gustado que el código en JavaScript únicamente se encargase de formar la vista. Es decir que el cliente solo recibiera los datos para formar la vista.

Otra de las cosas a mencionar son las transacciones. Como ya he comentado, cada vez que es necesario escribir un dato dentro del contrato, es necesario pagar por él. En aplicaciones como *Twitter* o el *Diario* están bien planteadas, ya que por cada dato que quieras insertar, tenemos que pagar por él. Pero en aplicaciones como la del *tres en raya* por cada movimiento que haces tienes que pagar, estaría mejor poder realizar toda la transacción de una, de tal manera que pagues por la partida.

En definitiva, el desarrollo de aplicaciones en una Blockchain es una alternativa muy interesante a tener en cuenta. El hecho de poder tener un programa ejecutándose en un "servidor" que no pertenece a nadie permite muchas posibilidades. Como sabemos, nada es gratis y aquí se puede ver lo que podría ser un cambio en el modelo de negocio de las aplicaciones.

Repitiendo un caso práctico como antes para ponernos en situación, en la aplicación de *diario* el coste de publicar una entrada cuyo contenido es "Hola, ¡¡buenos días!!" sería 0.00383 ether que al cambio a euros en este momento es de 70 céntimos. Con lo que esto podría ser viable. También me gustaría resaltar que el precio del Ether, la moneda de Ethereum está sujeto al libre mercado que lo rige con lo que el precio en euros no es estable.

Como mensaje final hay que decir que creo que dentro de un tiempo cuando se estandarice, se podrán ver aplicaciones más interesantes con una funcionalidad más amplia y con una usabilidad hacia el usuario más amigable.

6.2 Líneas futuras

Como se ha podido apreciar en este proyecto, el estudio ha acotado todo un ámbito tan grande como Blockchain al desarrollo de casos de uso para una

plataforma. Generalizar cuáles serían próximos desarrollos sería un tanto especulativo, pero con la información adquirida con esta plataforma se puede elaborar algunas ideas.

En general cualquier aplicación que quiera beneficiarse de tener una estructura descentralizada, puede desarrollarse en Blockchain. Con lo visto en la plataforma de Ethereum se puede afirmar que aplicaciones tipo CRUD son una alternativa muy viable para funcionar correctamente. Dicho esto, por ejemplo, se podría realizar un proyecto que fuera un mercado online para vender tipo Ebay o aplicaciones similares. Estas no se verían afectadas por terceras entidades que custodien los datos que manejan.

Otro sector que puede verse beneficiado con este tipo de aplicaciones CRUD descentralizadas son entidades gubernamentales o sanitarias. Poniendo el caso de entidades sanitarias, los datos de los pacientes se verían guardados en una red general, la cual es accesible desde cualquier centro. Además, cada paciente es dueño de sus propios datos y estos podrían ser consultados de manera personal.

Finalmente quiero comentar que en cuanto al desarrollo de aplicaciones descentralizadas que no se basen en un CRUD, esto puede ser complicado debido a las limitaciones de Ethereum. La idea con la que se desarrolló Solidity fue con la creación de contratos inteligentes. Es decir, que implementar cierta lógica en las aplicaciones descentralizadas que no tenga que ver con contratos inteligentes puede no ser viable.

7 Bibliografía

[1] S. Haber y W. S. Stornetta, «Haber_Stornetta.pdf,» [En línea]. Available: https://www.anf.es/pdf/Haber_Stornetta.pdf.

[2] «/www.researchgate.net,» [En línea]. Available: https://www.researchgate.net/figure/Key-Elements-of-Blockchain-Systems_fig1_327711685.

[3] Bitcoin.org, «bitcoin.org,» [En línea]. Available: https://bitcoin.org/es/.

[4] S. Nakamoto, "bitcoin.org," [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[5] «en.bitcoinwiki.org,» [En línea]. Available: https://en.bitcoinwiki.org/wiki/Block.

[6] «bitcoin.org,» [En línea]. Available: https://bitcoin.org/en/developer-reference#block-headers.

[7] N. (. s. agency), «web.archive.org,» [En línea]. Available: https://web.archive.org/web/20160330153520/http://www.staff.science.uu.nl/~werkh108/docs/study/Y5_07_08/infocry/project/Cryp08.pdf.

[8] R. Merkle, «https://people.eecs.berkeley.edu,» [En línea]. Available: https://people.eecs.berkeley.edu/~raluca/cs261-f15/readings/merkle.pdf.

[9] N. Sabo, «http://www.truevaluemetrics.org,» [En línea]. Available: http://www.truevaluemetrics.org/DBpdfs/BlockChain/Nick-Szabo-Smart-Contracts-Building-Blocks-for-Digital-Markets-1996-14591.pdf.

[10] V. Buterin, «about.me,» [En línea]. Available: https://about.me/vitalik_buterin.

[11] Ethereum, «github.com,» [En línea]. Available: https://github.com/ethereum/solidity.

[12] entethalliance.org, «entethalliance.org,» [En línea]. Available: https://entethalliance.org/.

[13] Hyperledger, «hyperledger.org,» [En línea]. Available: https://www.hyperledger.org/use/fabric.

[14] Hyperledger, «hyperledger.org,» [En línea]. Available: https://www.hyperledger.org/use/sawtooth.

[15] Ripple, «https://ripple.com/,» [En línea]. Available: https://ripple.com/.

[16] Hedera, «www.hedera.com/,» [En línea]. Available: https://www.hedera.com/.

[17] Alastria, «alastria.io,» [En línea]. Available: https://alastria.io/wp-content/uploads/2020/05/20200520_Alastria-Corporate-presentation_ESP.pdf.

[18] BLUE, «crue.org,» [En línea]. Available: https://tic.crue.org/blue/.

[19] r3, «r3.com,» [En línea]. Available: https://www.r3.com/corda-platform/.

[20] CORDA, «corda.net,» [En línea]. Available: https://www.corda.net/.

[21] «Ganache,» [En línea]. Available: https://www.trufflesuite.com/ganache.

[22] Node.js, «github,» [En línea]. Available: https://github.com/nodesource/distributions/blob/master/README.md#debinstall.

[23] k. b. danfinlay, «addons.mozilla.org,» [En línea]. Available: https://addons.mozilla.org/es/firefox/addon/ether-metamask/.

[24] «Bootstrap,» [En línea]. Available: https://getbootstrap.com/docs/4.5/getting-started/download/.